

JBoss DNA

1

Reference Guide

0.3

by Randall M. Hauch

Target audience	v
1. Introduction to JBoss DNA	1
1.1. Use cases for JBoss DNA	1
1.2. What is metadata?	1
1.3. What is JCR?	2
1.4. Project roadmap	4
1.5. Development methodology	4
1.6. JBoss DNA modules	5
2. Developer tools	9
2.1. JDK	9
2.2. Subversion	9
2.3. Maven	10
2.4. Continuous integration with Hudson	12
2.5. Eclipse IDE	12
2.6. Releasing	13
2.6.1. Building all artifacts and assemblies	14
2.6.2. Determine the version to be released	14
2.6.3. Release dry run	14
2.6.4. Prepare for the release	15
2.6.5. Perform the release	15
2.7. Summary	16
3. Class loaders	17
3.1. Class loader factory	17
3.2. Standard class loader factory	18
3.3. Maven Repository class loader factory	18
3.4. Summary	19
4. Environment	21
4.1. Security	21
4.1.1. Authenticating with JAAS	21
4.1.2. Authorization with JAAS	22
4.2. Execution contexts	22
4.3. Summary	25
5. Repositories	27
5.1. Repository connectors	27
5.2. Repository Service	30
5.3. Out-of-the-box repository connectors	33
5.3.1. In-memory connector	33
5.3.2. JBoss Cache connector	34
5.3.3. Federating connector	36
5.4. Writing custom connectors	40
5.4.1. Creating the Maven 2 project	41
5.4.2. Implementing a <code>RepositorySource</code>	43
5.4.3. Implementing a <code>RepositoryConnection</code>	46
5.4.4. Testing custom connectors	57

5.4.5. Configuring and deploying custom connectors	57
5.5. Graph API for using connectors	57
5.6. Summary	61
6. Content Repositories for Java (JCR)	63
6.1. Obtaining JCR repositories	63
6.2. Creating JCR sessions	63
6.3. Summary	65
7. Sequencing content	67
7.1. Sequencing Service	67
7.2. Sequencer Configurations	69
7.2.1. Path Expressions	71
7.3. JBoss DNA Sequencers	73
7.3.1. Image sequencer	73
7.3.2. Microsoft Office document sequencer	74
7.3.3. ZIP archive sequencer	75
7.3.4. Java source sequencer	76
7.3.5. MP3 audio file sequencer	76
7.3.6. JCR Compact Node Definition (CND) file sequencer	77
7.4. Creating custom sequencers	77
7.4.1. Creating the Maven 2 project	77
7.4.2. Implementing the StreamSequencer interface	81
7.4.3. Testing custom sequencers	85
7.4.4. Deploying custom sequencers	87
7.5. Summary	87
8. MIME types	89
8.1. JBoss DNA MIME type detectors	89
8.1.1. Aperture MIME type detector	90
8.2. Writing custom detectors	90
8.2.1. Creating the Maven 2 project	91
9. Configuration	95
10. Testing	97
10.1. Unit tests	97
10.2. Integration tests	97
10.3. Writing tests	98
11. Looking to the future	101

Target audience

This reference guide is for the developers of JBoss DNA and those users that want to have a better understanding of how JBoss DNA works or how to extend the functionality. For a higher-level introduction to JBoss DNA, see the [Getting Started](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/manuals/gettingstarted/html/index.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/manuals/gettingstarted/html/index.html] document.

If you have any questions or comments, please feel free to contact JBoss DNA's [user mailing list](mailto:dna-users@jboss.org) [mailto:dna-users@jboss.org] or use the [user forums](http://www.jboss.com/index.html?module=bb&op=viewforum&f=272) [http://www.jboss.com/index.html?module=bb&op=viewforum&f=272]. If you'd like to get involved on the project, join the [mailing lists](http://www.jboss.org/dna/lists.html) [http://www.jboss.org/dna/lists.html], [download the code](http://www.jboss.org/dna/subversion.html) [http://www.jboss.org/dna/subversion.html] and get it building, and visit our [JIRA issue management system](http://jira.jboss.org/jira/browse/DNA) [http://jira.jboss.org/jira/browse/DNA]. If there's something in particular you're interested in, talk with the community - there may be others interested in the same thing.

Introduction to JBoss DNA

The JBoss DNA project is building a unified metadata repository system that is *JCR-compliant* and capable of federating information from a variety of back-end systems. To client applications, JBoss DNA looks and behaves like a regular JCR repository that they search, navigate, version, and listen for changes. But under the covers, JBoss DNA gets its content by federating multiple back-end systems (like databases, services, other repositories, etc.), allowing those systems to continue "owning" the information while ensuring the unified repository stays up-to-date and in sync. JBoss DNA also analyzes the files you upload into the repository and turns it into information you can use more effectively.

This document goes into detail about JBoss DNA and its capabilities, features, architecture, components, extension points, security, configuration, and the development environment and procedures. So whether you're a developer on the project, or you're trying to learn the intricate details of how JBoss DNA works, this document hopefully serves a good reference for developers on the project.

1.1. Use cases for JBoss DNA

JBoss DNA repositories can be used in a variety of applications. One of the more obvious use cases for a metadata repository is in provisioning and management, where it's critical to understand and keep track of the metadata for models, database, services, components, applications, clusters, machines, and other systems used in an enterprise. Governance takes that a step farther, by also tracking the policies and expectations against which performance of the systems described by the repository can be verified. In these cases, a repository is an excellent mechanism for managing this complex and highly-varied information.

But these large and complex use cases aren't the only way to use a JBoss DNA repository. You could use an embedded JBoss DNA repository to manage configuration information for an application, or you could use JBoss DNA just provide a JCR interface on top of a few non-JCR systems.

The point is that JBoss DNA can be used in many different ways, ranging from the very tiny embedded repository to a large and distributed enterprise-grade repository. The choice is yours.

1.2. What is metadata?

Before we dive into more detail about JBoss DNA and metadata repositories, it's probably useful to explain what we mean by the term "metadata." Simply put, *metadata* is the information you need to manage something. For example, it's the information needed to configure an operating system, or the description of the information in an LDAP tree, or the topology of your network. It's the configuration of an application server or enterprise service bus. It's the steps involved in validating an application before it can go into production. It's the description of your database schemas, or of your services, or of the messages going in and coming out of a service. JBoss DNA is designed to be a repository for all this (and more).

There are a couple of important things to understand about metadata. First, many systems manage (and frequently change) their own metadata and information. Databases, applications, file systems, source code management systems, services, content management systems, and even other repositories are just a few types of systems that do this. We can't pull the information out and duplicate it, because then we risk having multiple copies that are out-of-sync. Ideally, we could access all of this information through a homogenous API that also provides navigation, caching, versioning, search, and notification of changes. That would make our lives significantly easier.

What we want is [federation](#). We can connect to these back-end systems to dynamically access the content and project it into a single, unified repository. We can also cache it for faster access, as long as the cache can be invalidated based upon time or event. But we also need to maintain a clear picture of where all the bits come from, so users can be sure they're looking at the right information. And we need to make it as easy as possible to write new connectors, since there are a lot of systems out there that have information we want to federate.

The second important characteristic of the metadata is that a lot of it is represented as files, and there are a lot of different file formats. These include source code, configuration files, web pages, database schemas, XML schemas, service definitions, policies, documents, spreadsheets, presentations, images, audio files, workflow definitions, business rules, and on and on. And logically if files contain metadata, we want to add those files to our metadata repository. The problem is, all that metadata is tied up as blobs in the repository. Ideally, our repository would automatically extract from those files the content that's most useful to us, and place that content inside the repository where it can be much more easily used, searched, related, and analyzed. JBoss DNA does exactly this via a process we call [sequencing](#), and it's an important part of a metadata repository.

The third important characteristic of metadata is that it rarely stays the same. Different consumers of the information need to see different views of it. Metadata about two similar systems is not always the same. The metadata often needs to be tagged or annotated with additional information. And the things being described often change over time, meaning the metadata has to change, too. As a result, the way in which we store and manage the metadata has to be flexible and able to adapt to our ever-changing needs, and the object model we use to interact with the repository must accommodate these needs. The graph-based nature of the JCR API provides this flexibility while also giving us the ability to constrain information when it needs to be constrained.

1.3. What is JCR?

There are a lot of choices for how applications can store information persistently so that it can be accessed at a later time and by other processes. The challenge developers face is how to use an approach that most closely matches the needs of their application. This choice becomes more important as developers choose to focus their efforts on application-specific logic, delegating much of the responsibilities for persistence to libraries and frameworks.

Perhaps one of the easiest techniques is to simply store information in *files*. The Java language makes working with files relatively easy, but Java really doesn't provide many bells and whistles. So using files is an easy choice when the information is either not complicated (for example

property files), or when users may need to read or change the information outside of the application (for example log files or configuration files). But using files to persist information becomes more difficult as the information becomes more complex, as the volume of it increases, or if it needs to be accessed by multiple processes. For these situations, other techniques often have more benefits.

Another technique built into the Java language is *Java serialization*, which is capable of persisting the state of an object graph so that it can be read back in at a later time. However, Java serialization can quickly become tricky if the classes are changed, and so it's beneficial usually when the information is persisted for a very short period of time. For example, serialization is sometimes used to send an object graph from one process to another. Using serialization for longer-term storage of information is more risky.

One of the more popular and widely-used persistence technologies is the *relational database*. Relational database management systems have been around for decades and are very capable. The Java Database Connectivity (JDBC) API provides a standard interface for connecting to and interacting with relational databases. However, it is a low-level API that requires a lot of code to use correctly, and it still doesn't abstract away the DBMS-specific SQL grammar. Also, working with relational data in an object-oriented language can feel somewhat unnatural, so many developers map this data to classes that fit much more cleanly into their application. The problem is that manually creating this mapping layer requires a lot of repetitive and non-trivial JDBC code.

Object-relational mapping libraries automate the creation of this mapping layer and result in far less code that is much more maintainable with performance that is often as good as (if not better than) handwritten JDBC code. The new *Java Persistence API (JPA)* [<http://java.sun.com/developer/technicalArticles/J2EE/jpa/>] provide a standard mechanism for defining the mappings (through annotations) and working with these entity objects. Several commercial and open-source libraries implement JPA, and some even offer additional capabilities and features that go beyond JPA. For example, *Hibernate* [<http://www.hibernate.org>] is one of the most feature-rich JPA implementations and offers object caching, statement caching, extra association mappings, and other features that help to improve performance and usefulness. Plus, Hibernate is open-source (with support offered by *JBoss* [<http://www.jboss.com>]).

While relational databases and JPA are solutions that work well for many applications, they are more limited in cases when the information structure is highly flexible, the structure is not known *a priori*, or that structure is subject to frequent change and customization. In these situations, *content repositories* may offer a better choice for persistence. Content repositories are almost a hybrid with the storage capabilities of relational databases and the flexibility offered by other systems, such as using files. Content repositories also typically provide other capabilities as well, including versioning, indexing, search, access control, transactions, and observation. Because of this, content repositories are used by content management systems (CMS), document management systems (DMS), and other applications that manage electronic files (e.g., documents, images, multi-media, web content, etc.) and metadata associated with them (e.g., author, date, status, security information, etc.). The *Content Repository for Java technology API* [<http://www.jcp.org/en/jsr/detail?id=170>] provides a standard Java API for working with content repositories. Abbreviated "JCR", this API was developed as part of the Java Community Process under *JSR-170* [<http://>

www.jcp.org/en/jsr/detail?id=170] and is being revised under [JSR-283](http://www.jcp.org/en/jsr/detail?id=283) [<http://www.jcp.org/en/jsr/detail?id=283>].

The *JBoss DNA project* is building unified metadata repository system that is compliant with JCR. Nearly all of these capabilities are to be hidden below the JCR API and involve automated processing of the information in the repository. Thus, JBoss DNA can add value to existing repository implementations. For example, JCR repositories offer the ability to upload files into the repository and have the file content indexed for search purposes. JBoss DNA also defines a library for "sequencing" content - to extract meaningful information from that content and store it in the repository, where it can then be searched, accessed, and analyzed using the JCR API.

JBoss DNA has other features as well. You can create federated repositories that dynamically merge the information from multiple databases, services, applications, and other JCR repositories. JBoss DNA also will allow you to create customized views based upon the type of data and the role of the user that is accessing the data. And yet another is to create a REST-ful API to allow the JCR content to be accessed easily by other applications written in other languages.

1.4. Project roadmap

The roadmap for JBoss DNA is managed in the project's [JIRA instance](http://jira.jboss.org/jira/browse/DNA) [<http://jira.jboss.org/jira/browse/DNA>] . The roadmap shows the different tasks, requirements, issues and other activities that have been targeted to each of the upcoming releases. (The [roadmap report](http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel) [<http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel>] always shows the next three releases.)

By convention, the JBoss DNA project team periodically review JIRA issues that aren't targeted to a release, and then schedule them based upon current workload, severity, and the roadmap. And if we review an issue and don't know how to target it, we target it to the [Future Releases](http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel) [<http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel>] bucket.

At the start of a release, the project team reviews the roadmap, identifies the goals for the release, and targets (or retargets) the issues appropriately.

1.5. Development methodology

Rather than use a single formal development methodology, the JBoss DNA project incorporates those techniques, activities, and processes that are practical and work for the project. In fact, the committers are given a lot of freedom for how they develop the components and features they work on.

Nevertheless, we do encourage familiarity with several major techniques, including:

- [Agile software development](http://en.wikipedia.org/wiki/Agile_software_development) [http://en.wikipedia.org/wiki/Agile_software_development] includes those software methodologies (e.g., Scrum) that promote development iterations and open collaboration. While the JBoss DNA project doesn't follow these closely, we do emphasize

the importance of always having running software and using running software as a measure of progress. The JBoss DNA project also wants to move to more frequent releases (on the order of 4-6 weeks)

- **Test-driven development (TDD)** [http://en.wikipedia.org/wiki/Test-driven_development] techniques encourage first writing test cases for new features and functionality, then changing the code to add the new features and functionality, and finally the code is refactored to clean-up and address any duplication or inconsistencies.
- **Behavior-driven development (BDD)** [<http://behaviour-driven.org/>] is an evolution of TDD, where developers specify the desired behaviors first (rather than writing "tests"). In reality, this BDD adopts the language of the user so that tests are written using words that are meaningful to users. With recent test frameworks (like JUnit 4.4), we're able to write our unit tests to express the desired behavior. For example, a test class for sequencer implementation might have a test method `shouldNotThrowAnErrorWhenStreamIsNull()`, which is very easy to understand the intent. The result appears to be a larger number of finer-grained test methods, but which are more easily understood and easier to write. In fact, many advocates of BDD argue that one of the biggest challenges of TDD is knowing what tests to write in the beginning, whereas with BDD the shift in focus and terminology make it easier for more developers to enumerate the tests they need.
- **Lean software development** [http://en.wikipedia.org/wiki/Lean_software_development] is an adaptation of [lean manufacturing techniques](http://en.wikipedia.org/wiki/Lean_manufacturing) [http://en.wikipedia.org/wiki/Lean_manufacturing], where emphasis is placed on eliminating waste (e.g., defects, unnecessary complexity, unnecessary code/functionality/features), delivering as fast as possible, deferring irrevocable decisions as much as possible, continuous learning (continuously adapting and improving the process), empowering the team (or community, in our case), and several other guidelines. Lean software development can be thought of as an evolution of agile techniques in the same way that behavior-driven development is an evolution of test-driven development. Lean techniques help the developer to recognize and understand how and why features, bugs, and even their processes impact the development of software.

1.6. JBoss DNA modules

JBoss DNA consists of the following modules:

- **dna-common** is a low-level library of common utilities and frameworks, including logging, progress monitoring, internationalization/localization, text translators, component management, and class loader factories.
- **dna-graph** defines the graph Application Programming Interface (API) and Service Provider Interface (SPI) for DNA, including the repository connectors, sequencers, graph interfaces, and MIME type detectors.
- **dna-repository** is the main module and provides the repository-oriented services, including the Repository Service, Sequencing Service, Observation Service, and Rules Service.

- **dna-jcr** provides the JBoss DNA implementation of the JCR API, which relies upon a repository connector, such as the Federation Connector (see `dna-connector-federation`).
- **dna-integration-tests** provides a home for all of the integration tests that involve more components than just unit tests. Integration tests are often more complicated, take longer, and involve testing the integration and functionality of many components (whereas unit tests focus on testing a single class or component and may use stubs or mock objects for other components).

The following modules are optional extensions that may be used selectively and as needed (and are located in the source under the `extensions/` directory):

- **dna-maven-classloader** is a small library that provides a `ClassLoaderFactory` implementation that can create `java.lang.ClassLoader` instances capable of loading classes given a Maven Repository and a list of Maven coordinates. The Maven Repository can be managed within a JCR repository.
- **dna-connector-federation** is a DNA repository connector that federates, integrates and caches information from multiple sources (via other repository connectors).
- **dna-connector-inmemory** is a simple DNA repository connector that manages content within memory. This can be used as a simple cache or as a transient repository.
- **dna-connector-jboss-cache** is a DNA repository connector that manages content within a [JBoss Cache](http://www.jboss.org/jboss-cache/) [http://www.jboss.org/jboss-cache/] instance. JBoss Cache is a powerful cache implementation that can serve as a distributed cache and that can persist information. The cache instance can be found via JNDI or created and managed by the connector.
- **dna-connector-svn** is a prototype DNA sequencer that obtains content from a Subversion repository, providing that content in the form of `nt:file` and `nt:folder` nodes. *This is still under development.*
- **dna-sequencer-zip** is a DNA sequencer that extracts from ZIP archives the files (with content) and folders.
- **dna-sequencer-xml** is a DNA sequencer that extracts the structure and content from XML files.
- **dna-sequencer-images** is a DNA sequencer that extracts the image metadata (e.g., size, date, etc.) from PNG, JPEG, GIF, BMP, PCS, IFF, RAS, PBM, PGM, and PPM image files.
- **dna-sequencer-mp3** is a DNA sequencer that extracts metadata (e.g., author, album name, etc.) from MP3 audio files.
- **dna-sequencer-java** is a DNA sequencer that extracts the package, class/type, member, documentation, annotations, and other information from Java source files.
- **dna-sequencer-msoffice** is a DNA sequencer that extracts metadata and summary information from [Microsoft Office](http://office.microsoft.com/en-us/) [http://office.microsoft.com/en-us/] documents. For example, the sequencer extracts from a PowerPoint presentation the outline as well as thumbnails of each slide. Microsoft Word and Excel files are also supported.

- **dna-sequencer-cnd** is a prototype DNA sequencer that extracts JCR node definitions from JCR Compact Node Definition (CND) files. *This is still under development.*
- **dna-sequencer-jbpm-jpdl** is a prototype DNA sequencer that extracts process definition metadata from jBPM process definition language (jPDL) files. *This is still under development.*
- **dna-mimetype-detector-aperture** is a DNA MIME type detector that uses the [Aperture](http://aperture.sourceforge.net/) [http://aperture.sourceforge.net/] library to determine the best MIME type from the filename and file contents.

There are also documentation modules (located in the source under the `docs/` directory):

- **docs-getting-started** is the project with the [DocBook](http://www.docbook.org/) [http://www.docbook.org/] source for the JBoss DNA Getting Started document.
- **docs-getting-started-examples** is the project with the Java source for the example application used in the JBoss DNA Getting Started document.
- **docs-reference-guide** is the project with the [DocBook](http://www.docbook.org/) [http://www.docbook.org/] source for this document, the JBoss DNA Reference Guide document.

Finally, there is a module that represents the whole JBoss DNA project:

- **dna** is the parent project that aggregates all of the other projects and that contains some asset files to create the necessary Maven artifacts during a build.

Each of these modules is a Maven project with a group ID of `org.jboss.dna`. All of these projects correspond to artifacts in the [JBoss Maven 2 Repository](http://repository.jboss.com/maven2/) [http://repository.jboss.com/maven2/].

Developer tools

The JBoss DNA project uses [Maven](#) as its primary build tool, [Subversion](#) for its source code repository, [JIRA](#) for the issue management and bug tracking system, and [Hudson](#) for the continuous integration system. We do not stipulate a specific integrated development environment (IDE), although most of us use [Eclipse](#) and rely upon the code formatting and compile preferences to ensure no warnings or errors.

The rest of this chapter talks in more detail about these different tools and how to set them up.

2.1. JDK

Currently, JBoss DNA is developed and built using [JDK 5](#) [http://java.sun.com/javase/downloads/index_jdk5.jsp]. So if you're trying to get JBoss DNA to compile locally, you should make sure you have the JDK 5 installed and are using it. If you're a contributor, you should make sure that you're using JDK 5 before committing any changes.



Note

You should be able to use the [latest JDK](#) [<http://java.sun.com/javase/downloads/index.jsp>], which is currently JDK 6. We periodically try to build JBoss DNA using JDK 6, but it's not our official JDK (yet).

Why do we build using JDK 5 and not 6? The main reason is that if we were to use JDK 6, then JBoss DNA couldn't really be used in any applications or projects that still used JDK 5. Plus, anybody using JDK 6 can still use JBoss DNA. However, considering that the end-of-life for Java 5 is [October 2009](#) [<http://java.sun.com/products/archive/eol.policy.html>], we may be switching to Java 6 sometime in 2009.

When installing a JDK, simply follow the procedure for your particular platform. On most platforms, this should set the `JAVA_HOME` environment variable. But if you run into any problems, first check that this environment variable was set to the correct location, and then check that you're running the version you expect by running the following command:

```
$ java -version
```

If you don't see the correct version, double-check your JDK installation.

2.2. Subversion

JBoss DNA uses Subversion as its source code management system, and specifically the instance at [JBoss.org](#) [<http://www.jboss.org>]. Although you can view the [trunk](#) [<http://anonsvn.jboss.org/>]

repos/dna/trunk/] of the Subversion repository directly (or using [FishEye](http://fisheye.jboss.org/browse/DNA/trunk/) [http://fisheye.jboss.org/browse/DNA/trunk/]) through your browser, in order to get more than just a few files of the latest version of the source code, you probably want to have an SVN client installed. Several IDE's have SVN support included (or available as plugins), but having the command-line SVN client is recommended. See <http://subversion.tigris.org/> for downloads and instructions for your particular platform.

Here are some useful URLs for the JBoss DNA Subversion:

Table 2.1. SVN URLs for JBoss DNA

Repository	URL
Anonymous Access URL	http://anonsvn.jboss.org/repos/dna/trunk/
Secure Developer Access URL	http://fisheye.jboss.org/browse/DNA/trunk/
FishEye Code Browser	https://svn.jboss.org/repos/dna/trunk/

2.3. Maven

JBoss DNA uses Maven 2 for its build system, as is this example. Using Maven 2 has several advantages, including the ability to manage dependencies. If a library is needed, Maven automatically finds and downloads that library, plus everything that library needs. This means that it's very easy to build the examples - or even create a maven project that depends on the JBoss DNA JARs.

To use Maven with JBoss DNA, you'll need to have [JDK 5 or 6](#) and Maven 2.0.9 (or higher).

Maven can be downloaded from <http://maven.apache.org/>, and is installed by unzipping the maven-2.0.9-bin.zip file to a convenient location on your local disk. Simply add \$MAVEN_HOME/bin to your path and add the following profile to your ~/.m2/settings.xml file:

```
<settings>
  <profiles>
    <profile>
      <id>jboss.repository</id>
      <activation>
        <property>
          <name>!jboss.repository.off</name>
        </property>
      </activation>
      <repositories>
        <repository>
          <id>snapshots.jboss.org</id>
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
```



```

        <enabled>true</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>repository.jboss.org</id>
      <url>http://repository.jboss.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>repository.jboss.org</id>
      <url>http://repository.jboss.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
    <pluginRepository>
      <id>snapshots.jboss.org</id>
      <url>http://snapshots.jboss.org/maven2</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
</profiles>
</settings>

```

This profile informs Maven of the two JBoss repositories ([snapshots](http://repository.jboss.org/maven2) [http://repository.jboss.org/maven2] and [releases](http://snapshots.jboss.org/maven2) [http://snapshots.jboss.org/maven2]) that contain all of the JARs for JBoss DNA and all dependent libraries.



Note

It is a policy of the project that the *source code and JARs* for *all* dependencies *must* be loaded into the JBoss repository. This is so that the project can always be built and that all source code is always available.

For more information about the JBoss Maven repository, see the [JBoss.org Wiki](http://wiki.jboss.org/wiki/Maven) [http://wiki.jboss.org/wiki/Maven].

There are just a few commands that are useful for building JBoss DNA (and it's [subprojects](#)). Usually, these are issued while at the top level of the code (usually just below `trunk/`), although issuing them inside a subproject just applies to that subproject.

Table 2.2. Useful Maven commands

Command	Description
<code>mvn clean</code>	Clean up all built artifacts (e.g., the <code>target/</code> directory in each project)
<code>mvn clean install</code>	Clean up all built artifacts, then compile, run the unit tests, and install the resulting JAR artifact(s) into your local Maven repository (e.g, usually <code>~/.m2/repository</code>).

2.4. Continuous integration with Hudson

JBoss DNA's continuous integration is done with several Hudson jobs on [JBoss.org](http://www.jboss.org) [http://www.jboss.org]. These jobs run periodically and basically run the Maven build process. Any build failures or test failures are reported, as are basic statistics and history for each job.

Table 2.3. Continuous integration jobs

Job	Description
Continuous on JDK 5 [http://hudson.jboss.org/hudson/job/DNA%20continuous%20on%20JDK1.5/]	Continuous build that runs after changes are committed to SVN. SVN is polled every 15 minutes.
Nightly on JDK 5 [http://hudson.jboss.org/hudson/job/DNA%20nightly%20integration%20on%20JDK1.5/]	Build that runs every night (usually around 2 a.m. EDT), regardless of whether changes have been committed to SVN since the previous night.

2.5. Eclipse IDE

Many of the JBoss DNA committers use the Eclipse IDE, and all project files required by Eclipse are committed in SVN, making it pretty easy to get an Eclipse workspace running with all of the JBoss DNA projects. Many of the JBoss DNA committers use the Eclipse IDE, and all project files required by Eclipse are committed in SVN, making it pretty easy to get an Eclipse workspace running with all of the JBoss DNA projects.

We're using the latest released version of Eclipse (3.4, called "Ganymede"), available from [Eclipse.org](http://www.eclipse.org) [http://www.eclipse.org/]. Simply follow the instructions for your platform.

After Eclipse is installed, create a new workspace. Before importing the JBoss DNA projects, import (via **File->Import->Preferences**) the subset of the Eclipse preferences by importing the `eclipse-preferences.epf` file (located under `trunk`). Then, open the Eclipse preferences and

open the **Java->Code Style-> Formatter** preference page, and press the "Import" button and choose the `eclipse-code-formatter-profile.xml` file (also located under `trunk`). This will load the code formatting preferences for the JBoss DNA project.

Then install Eclipse plugins for SVN and Maven. (Remember, you will have to restart Eclipse after installing them.) We use the following plugins:

Table 2.4. Continuous integration jobs

Eclipse Plugins	Update Site URLs
Subversive SVN Client	http://www.polarion.org/projects/subversive/download/eclipse/2.0/update-site/ http://www.polarion.org/projects/subversive/download/integrations/update-site/
Maven Integration for Eclipse	http://m2eclipse.sonatype.org/update/

After you check out the JBoss DNA codebase, you can import the JBoss DNA Maven projects into Eclipse as Eclipse projects. To do this, go to "File->Import->Existing Projects", navigate to the `trunk/` folder in the import wizard, and then check each of the [subprojects](#) that you want to have in your workspace. Don't forget about the projects under `extensions/` or `docs/`.

2.6. Releasing

This section outlines the basic process of releasing JBoss DNA. This **must** be done either by the project lead or only after communicating with the project lead.

Before continuing, your local workspace should contain no changes and should be a perfect reflection of Subversion. You can verify this by getting the latest from Subversion

```
$ svn update
```

and ensuring that you have no additional changes with

```
$ svn status
```

You may also want to note the revision number for use later on in the process. The release number is returned by the `svn update` command, but may also be found using

```
$ svn info
```

At this point, you're ready to verify that everything builds normally.

2.6.1. Building all artifacts and assemblies

By default, the project's Maven build process does *not* build the documentation, JavaDocs, or assemblies. These take extra time, and most of our builds don't require them. So the first step of releasing JBoss DNA is to use Maven to build all of regular artifacts (e.g., JARs) and these extra documents and assemblies.



Note

Before running Maven commands to build the releases, increase the memory available to Maven with this command: `$ export MAVEN_OPTS=-Xmx256m`

To perform this complete build, issue the following command while in the `target/` directory:

```
$ mvn -P assembly clean javadoc:javadoc install
```

This command runs the "clean", "javadoc:javadoc", and "install" goals using the "assembly" profile, which adds the production of JavaDocs, the Getting Started document, the Reference Guide document, the Getting Started examples, and several ZIP archives. The order of the goals is important, since the "install" goal attempts to include the JavaDoc in the archives.

After this build has completed, verify that the assemblies under `target/` have actually been created and that they contain the correct information. At this point, we know that the actual Maven build process is building everything we want and will complete without errors. We can now proceed with preparing for the release.

2.6.2. Determine the version to be released

The version being released should match the [JIRA](http://jira.jboss.org/jira/browse/DNA) [http://jira.jboss.org/jira/browse/DNA] road map. Make sure that all issues related to the release are closed. The project lead should be notified and approve that the release is taking place.

2.6.3. Release dry run

The next step is to ensure that all information in the POM is correct and contains all the information required for the release process. This is called a *dry run*, and is done with the Maven "release" plugin:

```
$ mvn -Passembly release:prepare -DdryRun=true
```

This may download a lot of Maven plugins if they already haven't been downloaded, but it will eventually prompt you for the release version of each of the Maven projects, the tag name for the

release, and the next development versions (again for each of the Maven projects). The default values are probably acceptable; if not, then check that the "`<version>`" tags in each of the POM files is correct and end with "-SNAPSHOT".

After the dry run completes you should clean up the files that the release plugin created in the dry run:

```
$ mvn -Passembly release:clean
```

2.6.4. Prepare for the release

Run the prepare step (without the `dryRun` option):

```
$ mvn -Passembly release:prepare
```

You will again be prompted for the release versions and tag name. These should be the same as what was used during the dry run. This will run the same steps as the dry run, with the additional step of tagging the release in SVN.

If there are any problems during this step, you should go back and try the dry run option.

2.6.5. Perform the release

Next run the perform step which will checkout the files from the tag, do a build, and deploy the generated artifacts.

```
$ mvn -Passembly release:perform
```

The deployment is done to the local file system (a local checkout of the JBoss Maven2 repository), so you will need to commit the new files after they are deployed. For more information, see the [JBoss wiki](http://wiki.jboss.org/wiki/Maven) [http://wiki.jboss.org/wiki/Maven].

Note that the release process updates your project's `pom.xml` files to change the "`<version>`" values to the next version. These will then need to be committed onto the trunk of SVN.

At this point, the software has been released and tagged, and it's been deployed to a local checked-out copy of the JBoss DNA Maven 2 repository (via the "`<distribution>`" section of the `pom.xml` files). Those need to be committed into the Maven 2 repository using SVN. And finally, the last thing is to publish the release onto the project's [downloads](http://www.jboss.org/dna/downloads.html) [http://www.jboss.org/dna/downloads.html] and [documentation](http://www.jboss.org/dna/docs) [http://www.jboss.org/dna/docs] pages.

2.7. Summary

In this chapter, we described the various aspects of developing code for the JBoss DNA project. Before we start talking about some of the details of JBoss DNA repositories, connectors, and sequencers, we'll first talk about some very ubiquitous information: how does JBoss DNA load all of the extension classes? This is the topic of the [next chapter](#).

Class loaders

JBoss DNA is designed around extensions: sequencers, connectors, MIME type detectors, and class loader factories. The core part of JBoss DNA is relatively small and has few dependencies, while all of the "interesting" components are extensions that plug into and are used by different parts of the core. The core doesn't really care what the extensions do or what external libraries they require, as long as the extension fulfills its end of the extension contract.

This means that you only need the core modules of JBoss DNA on the application classpath, while the extensions do not have to be on the application classpath. And because the core modules of JBoss DNA have few dependencies, the risk of JBoss DNA libraries conflicting with the application's are lower. Extensions, on the other hand, will likely have a lot of unique dependencies. By separating the core of JBoss DNA from the class loaders used to load the extensions, your application is isolated from the extensions and their dependencies. Of course, you can put all the JARs on the application classpath, too. (This is what the examples in the [Getting Started](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/manuals/gettingstarted/html/index.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/manuals/gettingstarted/html/index.html] document do.)

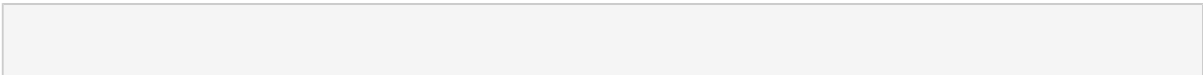
This design also allows you to select only those extensions that are interesting and useful for your application. Not every application needs all of the JBoss DNA functionality. Some applications may only need JBoss DNA sequencing, and specifically just a few types of sequencers. Other applications may not need sequencing but do want to use JBoss DNA federation capabilities.

Finally, the use of these formal extensions also makes it easier for you to write your own customized extensions. You may have proprietary file formats that you want to sequence. Or, you may have a non-JCR repository system that you want to access via JCR and maybe even federate with information from other sources. Since extensions do only one thing (e.g., be a sequencer, or a connector, etc.), its easier to develop those customizations.

3.1. Class loader factory

JBoss DNA loads all of the extension classes using class loaders returned by a *class loader factory*. Each time JBoss DNA wants to load a class, it needs the name of the class and an optional "class loader name". The meaning of the names is dependent upon the implementation of the class loader factory. For example, the [Maven class loader factory](#) expects the names to be [Maven coordinates](http://maven.apache.org/pom.html#Maven_Coordinates) [http://maven.apache.org/pom.html#Maven_Coordinates]. Either way, the class loader factory implementation uses the name to create and return a [ClassLoader](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html] instance that can be used to load the class. Of course, if no name is provided, then a JBoss DNA service just uses its class loader to load the class. (This is why putting all the extension jars on the classpath works.)

The class loader factory interface is pretty simple:



```
public interface ClassLoaderFactory {  
  
    /**  
     * Get a class loader given the supplied classpath. The meaning of the classpath is  
     * implementation-dependent.  
     * @param classpath the classpath to use  
     * @return the class loader; may not be null  
     */  
    ClassLoader   
    getClassLoader( String... classpath );  
}
```

In the [next chapter](#) we'll describe an [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) interface that is supplied to each of the JBoss DNA core services. This context interface actually extends the [ClassLoaderFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html) interface, so setting up an [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) implicitly sets up the class loader factory.

3.2. Standard class loader factory

JBoss DNA includes and uses as a default a standard class loader factory that just loads the classes using the Thread's current context class loader (if there is one), or a delegate class loader that defaults to the class loader that loaded the [StandardClassLoaderFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html) class. The class ignores any class loader names that are supplied.

3.3. Maven Repository class loader factory

The `dna-classloader-maven` project has a class loader factory implementation that parses the names into [Maven coordinates](http://maven.apache.org/pom.html#Maven_Coordinates), then uses those coordinates to look up artifacts in a Maven 2 repository. The artifact's POM file is used to determine the dependencies, which is done transitively to obtain the complete dependency graph. The resulting class loader has access to these artifacts in dependency order.

This class loader is also able to use a JCR repository that contains the equivalent contents of a Maven repository. However, JBoss DNA doesn't currently have any tooling to help populate that repository, so this component may be of limited use right now.

3.4. Summary

In this chapter, we described the framework used by JBoss DNA to load extension classes, like implementations of repositories, sequencers, MIME type detectors, and other components. [Next](#), we cover how JBoss security works and how the various components of JBoss DNA can access this security information as well as information about the environment in which the component is running.

Environment

The various components of JBoss DNA are designed as plain old Java objects, or POJOs. And rather than making assumptions about their environment, each component instead requires that any external dependencies necessary for it to operate must be supplied to it. This pattern is known as Dependency Injection, and it allows the components to be simpler and allows for a great deal of flexibility and customization in how the components are configured. And, JBoss DNA will soon provide a higher-level component that leverages the *JBoss Microcontainer* [<http://www.jboss.org/jbossmc>] to automatically assemble and wire together all the lower-level components.

4.1. Security

JBoss DNA uses the *Java Authentication and Authorization Service (JAAS)* [<http://java.sun.com/javase/technologies/security/>] for its security mechanism. Not only is this the standard approach for authenticating and authorizing in Java, but it also enables JBoss DNA to integrate existing security systems.

There are quite a few JAAS providers available, but one of the best and most powerful providers is *JBoss Security* [<http://www.jboss.org/jbosssecurity/>], which is the open source security framework used by JBoss. JBoss Security offers a number of JAAS login modules, including:

- **User-Roles Login Module** is a simple `javax.security.auth.login.LoginContext` implementation that uses usernames and passwords stored in a properties file.
 - **Client Login Module** prompts the user for their username and password.
 - **Database Server Login Module** uses a JDBC database to authenticate principals and associate them with roles.
 - **LDAP Login Module** uses an LDAP directory to authenticate principals. Two implementations are available.
 - **Certificate Login Module** authenticates using X509 certificates, obtaining roles from either property files or a JDBC database.
 - **Operating System Login Module** authenticates using the operating system's mechanism.
- and many others. Plus, JBoss Security also provides other capabilities, such as using XACML policies or using federated single sign-on. For more detail, see the *JBoss Security* [<http://www.jboss.org/jbosssecurity/>] project.

4.1.1. Authenticating with JAAS

JBoss DNA defers to JAAS to authenticate clients creating repository connections (known as JCR Sessions if using the JCR API), and generally expects the client application to obtain the JAAS *LoginContext* [<http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html>] or `.`. These are then passed to JBoss DNA, which merely verifies that authentication has been done.

As we'll see in the [next section](#), JBoss DNA has the notion of an [ExecutionContext](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html>] that can be created using JAAS login or access control contexts. These execution contexts therefore contain information about the subject.

We'll also see in the [chapter on JCR](#) how JAAS can be used for authentication when JCR Sessions are created.

4.1.2. Authorization with JAAS

JBoss DNA does not currently use any of the authorization features of JAAS. However, in future releases where authorization is supported, JBoss DNA will rely upon JAAS authorization just as it currently does for authentication.

4.2. Execution contexts

One of the objects that must be supplied to many JBoss DNA components is an [ExecutionContext](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html>]. Some components require this context to be passed into individual methods, allowing the context to vary with each method invocation. Other components require the context to be provided before it's used, and will use that context for all its operations (until it is given a different one).

What does an [ExecutionContext](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html>] represent? Quite simply, it's the set of objects that define the environment or context in which the method or component is currently operating. It includes a way for recording and reporting errors and problems. It includes the ability to [create class loaders](#) given a classpath of class loader names. It also includes information about the current [user](#). It includes access to factories that can be used to create and convert property values. And it includes factories for working with namespaces and fully-qualified names. In fact, as JBoss DNA evolves, more things may need to be added. Here is what the [ExecutionContext](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html>] interface looks like:

```
public interface ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] extends
    ClassLoaderFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html] {

    /**
     * Get the factories that should be used to create values for {@link Property properties}.
     * @return the property value factory; never null
     */
}
```

[ValueFactories](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/ValueFactories.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/ValueFactories.html] getValueFactories();

```
/**
 * Get the namespace registry for this context.
 * @return the namespace registry; never null
 */
```

[NamespaceRegistry](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/NamespaceRegistry.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/NamespaceRegistry.html] getNamespaceRegistry();

```
/**
 * Get the factory for creating {@link Property} objects.
 * @return the property factory; never null
 */
```

[PropertyFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PropertyFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PropertyFactory.html] getPropertyFactory();

```
/**
 * Get the current JAAS access control context.
 * @return the access control context; may be null
 */
```

[AccessControlContext](http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html] getAccessControlContext();

```
/**
 * Get the current JAAS login context.
 * @return the login context; may be null
 */
```

[LoginContext](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] getLoginContext();

```
/**
 * Get the JAAS subject for which this context was created.
 * @return the subject; never null
 */
```

[Subject](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html] getSubject();

```
/**
 * Return a logger associated with this context. This logger records only those activities within the
 * context and provide a way to capture the context-specific activities. All log messages are also
 * sent to the system logger, so classes that log via this mechanism should
```

<i>not</i>

also

```
 * {@link Logger#getLogger(Class)} obtain a system logger}.
```

```
* @param clazz the class that is doing the logging
* @return the logger, named after clazz; never null
*/
Logger [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/util/Logger.html] getLogger( Class<?> clazz );

/**
* Return a logger associated with this context. This logger records only those activities within the
* context and provide a way to capture the context-specific activities. All log messages are also
* sent to the system logger, so classes that log via this mechanism should
<i>not</i>
also
* {@link Logger#getLogger(Class)} obtain a system logger}.
* @param name the name for the logger
* @return the logger, named after clazz; never null
*/
Logger [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/util/Logger.html] getLogger( String name );
}
```

Notice that [*ExecutionContext*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] extends the [*ClassLoaderFactory*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html] interface described in the [previous chapter](#).

The fact that so many of the JBoss DNA components take [*ExecutionContext*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] instances gives us some interesting possibilities. For example, one execution context instance can be used as the highest-level (or "application-level") context for all of the services (e.g., [*RepositoryService*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html], [*SequencingService*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html], etc.). Then, an execution context could be created for each user that will be performing operations, and that user's context can be passed around to not only provide security information about the user but also to allow the activities being performed to be recorded for user feedback, monitoring and/or auditing purposes.

While execution contexts may sound complicated, they're actually very simple to use. In fact, JBoss DNA provides an factory interface for creating [*ExecutionContext*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] instances. Not surprisingly it's called [*ExecutionContextFactory*](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html] and it has methods for creating

contexts using [JAAS](#) login or access control contexts. JBoss DNA even provides a [BasicExecutionContextFactory](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/BasicExecutionContextFactory.html>] implementation that can be created using its no-arg constructor.

The following code fragment shows how easy it is to create various execution contexts:

```
ExecutionContextFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html] factory = new
    BasicExecutionContextFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/BasicExecutionContextFactory.html]();
ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] context1 = factory.create();

// Create a context for a user, authenticating using JAAS ...
CallbackHandler [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/callback/CallbackHandler.html] callbackHandler = ...
LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] loginContext = new LoginContext("username",callbackHandler);
ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] context2 = factory.create(loginContext);
```

These contexts (or the context factory) can then be passed to the various components as needed.

4.3. Summary

In this chapter, we covered security and environment topics as used throughout JBoss DNA. The [next chapter](#) will cover JBoss DNA repositories, including the connector framework, how DNA's JCR implementation works with connectors, what connectors are available (and how to use them), and how to write your own connector.

Repositories

There is a lot of information stored in many of different places: databases, repositories, SCM systems, registries, file systems, services, etc. The purpose of the federation engine is to allow applications to use the JCR API to access that information as if it were all stored in a single JCR repository, but to really leave the information where it is.

Why not just copy or move the information into a JCR repository? Moving it is probably pretty difficult, since most likely there are existing applications that rely upon that information being where it is. All of those applications would break or have to change. And copying the information means that we'd have to continually synchronize the changes. This not only is a lot of work, but it often makes it difficult to know whether information is accurate and "the master" data.

JBoss DNA lets us leave information where it, yet access it through the JCR API as if it were in one big repository. One major benefit is that existing applications that use the information in the original locations don't break, since they can keep using the information. But now our JCR clients can also access all the information, too. And if our federating JBoss DNA repository is configured to allow updates, JCR client applications can change the information in the repository and JBoss DNA will propagate those changes down to the original source, making those changes visible to all the other applications.

In short, all clients see the correct information, even when it changes in the underlying systems. But the JCR clients can get to all of the information in one spot, using one powerful standard API.

5.1. Repository connectors

With JBoss DNA, your applications use the [JCR API](http://www.jcp.org/en/jsr/detail?id=170) [http://www.jcp.org/en/jsr/detail?id=170] to work with the repository, but the DNA repository transparently fetches the information from different kinds of repositories and storage systems, not just a single purpose-built store. This is fundamentally what makes JBoss DNA different.

How does JBoss DNA do this? At the heart of JBoss DNA and it's JCR implementation is a simple graph-based *repository connector* system. Essentially, JBoss DNA's JCR implementation uses a single repository connector to access all content:

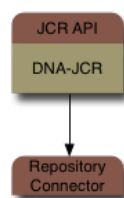


Figure 5.1. JBoss DNA's JCR implementation delegates to a repository connector

That single repository connector could use an in-memory repository, a JBoss Cache instance (including those that are clustered and replicated), or a federated repository where content from multiple sources is unified.

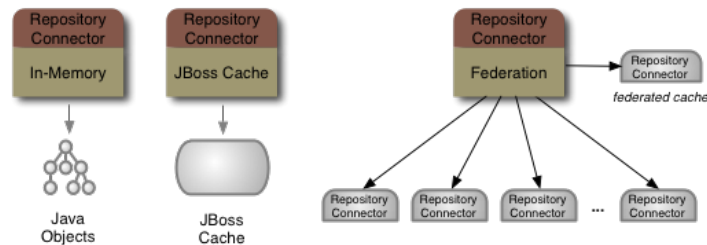


Figure 5.2. JBoss DNA can put JCR on top of multiple kinds of systems

Really, the federated connector gives us all kinds of possibilities, since we can use that connector on top of lots of connectors to other individual sources. This simple connector architecture is fundamentally what makes JBoss DNA so powerful and flexible. Along with a good library of connectors, which is what we're planning to create.

For instance, we want to build a connector to [other JCR repositories](#) [], and another that accesses the [local file system](#) []. We've already started on a [Subversion connector](#) [], which will allow JCR to access the files in a SVN repository (and perhaps push changes into SVN through a commit). And of course we want to create a connector that accesses [data](#) [] and [metadata](#) [] from relational databases. For more information, check out our [roadmap](http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel) [http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel]. Of course, if we don't have a connector to suit your needs, you can [write your own](#).

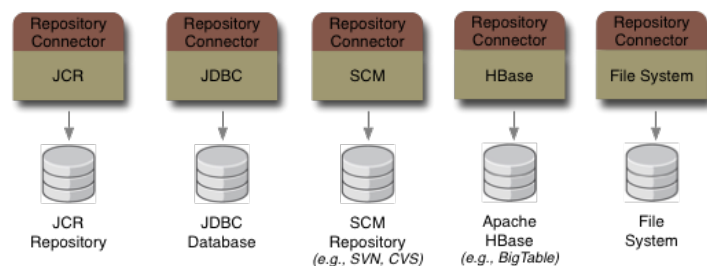


Figure 5.3. Future JBoss DNA connectors

It's even possible to put a different API layer on top of the connectors. For example, the new [New I/O \(JSR-203\)](#) [http://www.jcp.org/en/jsr/detail?id=203] API offers the opportunity to build new file system providers. This would be very straightforward to put on top of a JCR implementation, but it could be made even simpler by putting it on top of a DNA connector. In both cases, it'd be a trivial mapping from nodes that represent files and folders into JSR-203 files and directories, and events on those nodes could easily be translated into JSR-203 watch events. Then, simply choose a DNA connector and configure it to use the source you want to use.

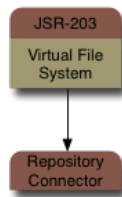


Figure 5.4. Virtual File System with JBoss DNA

Before we go further, let's define some terminology regarding connectors.

- A **connector** is the runnable code packaged in one or more JAR files that contains implementations of several interfaces (described below). A Java developer *writes* a connector to a type of source, such as a particular database management system, LDAP directory, source code management system, etc. It is then packaged into one or more JAR files (including dependent JARs) and deployed for use in applications that use JBoss DNA repositories.
- The description of a particular source system (e.g., the "Customer" database, or the company LDAP system) is called a **repository source**. JBoss DNA defines a [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] interface that defines methods describing the behavior and supported features and a method for establishing connections. A connector will have a class that implements this interface and that has JavaBean properties for all of the connector-specific properties required to fully describe an instance of the system. Use of JavaBean properties is not required, but it is highly recommended, as it enables reflective configuration and administration. Applications that use JBoss DNA create an instance of the connector's [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation and set the properties for the external source that the application wants to access with that connector.
- A repository source instance is then used to establish **connections** to that source. A connector provides an implementation of the [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] interface, which defines methods for interacting with the external system. In particular, the `execute(...)` method takes an [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] instance and a [Request](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] object. The object defines the environment in which the processing is occurring, including information about the JAAS [Subject](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html] and [LoginContext](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html]. The [Request](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] object describes the requested operations on the content, with different concrete subclasses representing each

type of activity. Examples of commands include (but not limited to) getting a node, moving a node, creating a node, changing a node, and deleting a node. And, if the repository source is able to participate in JTA/JTS distributed transactions, then the [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] must implement the `getXaResource()` method by returning a valid `javax.transaction.xa.XAResource` object that can be used by the transaction monitor.

As an example, consider that we want JBoss DNA to give us access through JCR to the schema information contained in a relational databases. We first have to develop a connector that allows us to interact with relational databases using JDBC. That connector would contain a `JdbcRepositorySource` Java class that implements [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html], and that has all of the various JavaBean properties for setting the name of the driver class, URL, username, password, and other properties. (Or we might have a JavaBean property that defines the JNDI name where we can find a JDBC `DataSource` instance pointing to our JDBC database.)

Our new connector would also have a `JdbcRepositoryConnection` Java class that implements the [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] interface. This class would probably wrap a JDBC database connection, and would implement the `execute(...)` method such that the nodes exposed by the connector describe the database schema of the database. For example, the connector might represent each database table as a node with the table's name, with properties that describe the table (e.g., the description, whether it's a temporary table), and with child nodes that represent each of the columns, keys and constraints.

To use our connector in an application that uses JBoss DNA, we need to create an instance of the `JdbcRepositorySource` for each database instance that we want to access. If we have 3 MySQL databases, 9 Oracle databases, and 4 PostgreSQL databases, then we'd need to create a total of 16 `JdbcRepositorySource` instances, each with the properties describing a single database instance. Those sources are then available for use by JBoss DNA components, including the [JCR](#) implementation.

So, we've so far learned what a repository connector is and how they're used to establish connections to the underlying sources and access the content in those sources. In the [next section](#), we'll show how these source instances can be configured, managed, and their connections pooled. After that, we'll look review JBoss DNA's [existing connectors](#) and show how to [create your own connectors](#).

5.2. Repository Service

The JBoss DNA [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] is the component that manages the *repository sources* and the connections to them. [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/) [http://www.jboss.org/file-access/default/members/dna/freezone/

[docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html)] instances can be programmatically added to the service, but the service can actually read its configuration from a configuration repository (which, by the way, is represented by a just another [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instance that's usually added programmatically to the service). The service connects to the configuration repository, reads the content in a particular area, and automatically sets up the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instances per the information found in the configuration repository.

The [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] also transparently maintains for each source a pool of reusable connections. The pooling properties can be controlled via the configuration repository, or adjusted programmatically.

Using a repository, then, involves simply asking the [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] for a [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] to the repository given the repository's name. If a source exists with that name, the service checks out a connection from the source's pool. The resulting connection is actually a wrapper around the underlying pooled connection, so the component that requested the connection can simply close it, and under the covers the actual connection is simply returned to the pool.

To instantiate the [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html], we need to first have a few other objects:

- A [ExecutionContextFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html] instance, as discussed [earlier](#).
- A [RepositoryLibrary](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html] instance that manages the list of [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instances, properly injects the execution contexts into each repository source, and provides a configurable pool of connections for each source.
- A *configuration repository* that contains descriptions of all of the repository sources as well as any information those sources need. Because this is a regular repository, this could be a simple repository with content loaded from an XML file, or it could be a shared central repository with information about all of the JBoss DNA repositories used across your organization.

With these components in place, we can then instantiate the [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html].

docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] and start it (using its [ServiceAdministrator](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html]). During startup, the service reads the configuration repository and loads any defined [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instances into the repository library, using the class loader factory (available in the [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html]) to obtain.

Here's sample code that shows how to set up and start the repository service. You can see something similar in the example application in the `startRepositories()` method of the `org.jboss.example.dna.repository.RepositoryClient` class.

```
// Create the factory for execution contexts, and create one ...
ExecutionContextFactory contextFactory =
    new BasicExecutionContextFactory();
ExecutionContext context = contextFactory.create();

// Create the library for the RepositorySource instances ...
RepositoryLibrary sources = new
RepositoryLibrary(contextFactory);

// Load into the source manager the repository source for the configuration repository ...
InMemoryRepositorySource configSource = new
InMemoryRepositorySource();
configSource.setName("Configuration");
sources.addSource(configSource);

// Now instantiate the Repository Service ...
RepositoryService service = new
RepositoryService(sources, configSource.getName(),
    context);
```

```
service.getAdministrator().start();
```

After startup completes, the repositories are ready to be used. The client application obtains the list of repositories and presents them to the user. When the user selects one, the client application starts navigating that repository starting at its root node (e.g., the "/" path). As you type a command to list the contents of the current node or to "change directories" to a different node, the client application obtains the information for the node using a simple procedure:

1. Get a connection to the repository.
2. Using the connection, find the current node and read its properties and children, putting the information into a simple Java plain old Java object (POJO).
3. Close the connection to the repository (in a finally block to ensure it always happens).

5.3. Out-of-the-box repository connectors

A number of repository connectors are already available in JBoss DNA, and are outlined in the following sections. Note that we do want to build [more connectors](https://jira.jboss.org/jira/secure/IssueNavigator.jspx?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311441) [https://jira.jboss.org/jira/secure/IssueNavigator.jspx?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311441] in the upcoming releases.

5.3.1. In-memory connector

The in-memory repository connector is a simple connector that creates a transient, in-memory repository. This repository is used as a very simple in-memory cache or as a standalone transient repository.

The [InMemoryRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/inmemory/InMemoryRepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/inmemory/InMemoryRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

Table 5.1. [InMemoryRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/inmemory/InMemoryRepositorySource.html)

[http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/inmemory/InMemoryRepositorySource.html] properties

Property	Description
name	The name of the repository source, which is used by the RepositoryService [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] when obtaining a RepositoryConnection [http://

Property	Description
	www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] by name.
jndiName	Optional property that, if used, specifies the name in JNDI where an InMemoryRepository [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/inmemory/InMemoryRepository.html] instance can be found. This is an advanced property that is infrequently used.
rootNodeUuid	Optional property that, if used, defines the UUID of the root node in the in-memory repository. If not used, then a new UUID is generated.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a RepositoryConnection [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.

5.3.2. JBoss Cache connector

The JBoss Cache repository connector allows a [JBoss Cache](http://www.jboss.org/jbosscache/) [http://www.jboss.org/jbosscache/] instance to be used as a JBoss DNA (and thus JCR) repository. This provides a repository that is an effective, scalable, and distributed cache, and is often paired with other repository sources to provide a local or [federated](#) repository.

The [JBossCacheSource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/jbosscache/JBossCacheSource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/jbosscache/JBossCacheSource.html] class provides a number of JavaBean properties that control its behavior:

Table

5.2.

[JBossCacheSource](#)

[<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/cache/properties>]

Property	Description
name	The name of the repository source, which is used by the RepositoryService [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/cache/repository/RepositoryService.html] when obtaining a RepositoryConnection [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/cache/graph/connectors/RepositoryConnection.html] by name.
cacheFactoryJndiName	Optional property that, if used, specifies the name in JNDI where an existing JBoss Cache Factory instance can be found. That factory would then be used if needed to create a JBoss Cache instance. If no value is provided, then the JBoss Cache <code>DefaultCacheFactory</code> class is used.
cacheConfigurationName	Optional property that, if used, specifies the name of the configuration that is supplied to the cache factory when creating a new JBoss Cache instance.
cacheJndiName	Optional property that, if used, specifies the name in JNDI where an existing JBoss Cache instance can be found. This should be used if your application already has a cache that is used, or if you need to configure the cache in a special way.
uuidPropertyName	Optional property that, if used, defines the property that should be used to find the UUID value for each node in the cache. "dna:uuid" is the default.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a RepositoryConnection [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/cache/graph/connectors/RepositoryConnection.html]

Property	Description
	to this source should be retried following a communication failure. The default value is '0'.
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.

5.3.3. Federating connector

The federated repository source provides a unified repository consisting of information that is dynamically federated from multiple other [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instances. This is a very powerful repository source that appears to be a single repository, when in fact the content is stored and managed in multiple other systems. Each [FederatedRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/FederatedRepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/FederatedRepositorySource.html] is typically configured with the name of another [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] that should be used as the local, unified cache of the federated content. The configuration also contains the names of the other [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instances that are to be federated along with the [Projection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/Projection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/Projection.html] definition describing where in the unified repository the content is to appear.

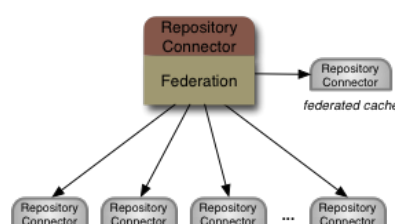


Figure 5.5. Federating multiple sources using the Federated Repository Connector

The federation connector works by effectively building up a single graph by querying each source and merging or unifying the responses. This information is cached, which improves performance, reduces the number of (potentially expensive) remote calls, reduces the load on the sources, and helps mitigate problems with source availability. As clients interact with the repository, this cache

is consulted first. When the requested portion of the graph (or "subgraph") is contained completely in the cache, it is returned immediately. However, if any part of the requested subgraph is not in the cache, each source is consulted for their contributions to that subgraph, and any results are cached.

This basic flow makes it possible for the federated repository to build up a local cache of the integrated graph (or at least the portions that are used by clients). In fact, the federated repository caches information in a manner that is similar to that of the Domain Name System (DNS). As sources are consulted for their contributions, the source also specifies whether it is the authoritative source for this information (some sources that are themselves federated may not be the information's authority), whether the information may be modified, the time-to-live (TTL) value (the time after which the cached information should be refreshed), and the expiration time (the time after which the cached information is no longer valid). In effect, the source has complete control over how the information it contributes is cached and used.

The federated repository also needs to incorporate *negative caching*, which is storage of the knowledge that something does *not* exist. Sources can be configured to contribute information only below certain paths (e.g., /A/B/C), and the federation engine can take advantage of this by never consulting that source for contributions to information on other paths. However, below that path, any negative responses must also be cached (with appropriate TTL and expiry parameters) to prevent the exclusion of that source (in case the source has information to contribute at a later time) or the frequent checking with the source.

The federated repository uses other [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html]s that are to be federated and a [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] that is to be used as the cache of the unified contents. These are configured in another [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] that is treated as a configuration repository. The [FederatedRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/FederatedRepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/FederatedRepositorySource.html] class uses JavaBean properties to define the name of the configuration repository and the path to the "dna:federation" node in that configuration repository containing the information about the cache and federated sources. This graph structure that is expected at this location is as follows:

```
<!-- Define the federation configuration. -->
<dna:federation xmlns:dna="http://www.jboss.org/dna"
  xmlns:jcr="http://www.jcp.org/jcr/1.0"
  dna:timeToCache="100000" >
  <!-- Define how the content in the 'Cache' source is to map to the federated cache -->
  <dna:cache>
    <dna:projection jcr:name="Cache" dna:projectionRules="/ => /" />
  </dna:cache>
</dna:federation>
```

```

</dna:cache>
<!-- Define how the content in the two sources maps to the federated/unified repository.
      This example puts the 'Cars' and 'Aircraft' content underneath '/vehicles', but the
      'Configuration' content (which is defined by this file) will appear under '/'. -->
<dna:projections>
  <dna:projection jcr:name="Cars" dna:projectionRules="/Vehicles => /" />
  <dna:projection jcr:name="Aircraft" dna:projectionRules="/Vehicles => /" />
  <dna:projection jcr:name="Configuration" dna:projectionRules="/" => /" />
</dna:projections>
</dna:federation>

```



Note

We're using XML to represent a graph structure, since the two map pretty well. Each XML element represents a node and XML attributes represent properties on a node. The name of the node is defined by either the `jcr:name` attribute (if it exists) or the name of the XML element. And we use XML namespaces to define the namespaces used in the node and property names. BTW, this is exactly how the XML graph importer works.

Notice that there is a cache projection and three source projections, and each projection defines one or more *projection rules* that are of the form:

```
pathInFederatedRepository => pathInSourceRepository
```

So, a projection rule `/Vehicles => /` projects the entire contents of the source so that it appears in the federated repository under the `"/Vehicles"` node.

The [FederatedRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/federation/FederatedRepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/federation/FederatedRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

Table 5.3. [FederatedRepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/federation/FederatedRepositorySource.html)

[http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/federation/FederatedRepositorySource.html] class provides a number of JavaBean properties

Property	Description
name	The name of the repository source, which is used by the RepositoryService [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/federation/RepositoryService.html]

Property	Description
	dna/repository/RepositoryService.html] when obtaining a RepositoryConnection [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] by name.
repositoryName	The name for the federated repository.
configurationSourceName	The name of the RepositorySource [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] that should be used as the configuration repository, and in which is defined how this federated repository is to be set up and configured. This name is supplied to the RepositoryConnectionFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html] that is provided to this instance when added to the RepositoryLibrary [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html].
configurationSourcePath	The path to the node in the configuration repository below which a "dna:federation" node exists with the graph structure describing how this federated repository is to be configured.
securityDomain	Optional property that, if used, specifies the name of the JAAS application context that should be used to establish the execution context for this repository. This should correspond to the JAAS login configuration located within the JAAS login configuration file, and should be used only if a "username" property is defined.
username	Optional property that, if used, defines the name of the JAAS subject that should be used to establish the execution context for

Property	Description
	this repository. This should be used if a "securityDomain" property is defined.
password	Optional property that, if used, defines the password of the JAAS subject that should be used to establish the <i>execution context</i> for this repository. If the password is not provided but values for the "securityDomain" and "username" properties are, then authentication will use the default JAAS callback handlers.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a <i>RepositoryConnection</i> [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.

5.4. Writing custom connectors

There may come a time when you want to tackle creating your own repository connector. Maybe the connectors we provide out-of-the-box don't work with your source. Maybe you want to use a different cache system. Maybe you have a system that you want to make available through a JBoss DNA repository. Or, maybe you're a contributor and want to help us round out our library with a new connector. No matter what the reason, creating a new connector is pretty straightforward, as we'll see in this section.

Creating a custom connector involves the following steps:

1. Create a Maven 2 project for your connector;
2. Implement the *RepositorySource* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html>] interface, using JavaBean properties for each bit of information the implementation will need to establish a connection to the source system.

Then, implement the [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] interface with a class that represents a connection to the source. The `execute(ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html], Request [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html])` method should process any and all requests that may come down the pike, and the results of each request can be put directly on that request.

Don't forget unit tests that verify that the connector is doing what it's expected to do. (If you'll be committing the connector code to the JBoss DNA project, please ensure that the unit tests can be run by others that may not have access to the source system. In this case, consider writing integration tests that can be easily configured to use different sources in different environments, and try to make the failure messages clear when the tests can't connect to the underlying source.)

3. Configure JBoss DNA to use your connector. This may involve just registering the source with the [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html], or it may involve adding a source to a configuration repository used by the [federated repository](#).
4. Deploy the JAR file with your connector (as well as any dependencies), and make them available to JBoss DNA in your application.

Let's go through each one of these steps in more detail.

5.4.1. Creating the Maven 2 project

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already [set up to use Maven](#), using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.



Note

JBoss DNA may provide in the future a Maven archetype for creating connector projects. If you'd find this useful and would like to help create it, please [join the community](#).

In lieu of a Maven archetype, you may find it easier to start with a small existing connector project. The **dna-connector-inmemory** project is small, but it may be tough to separate the stuff that every connector needs from the extra code and data structures that manage the content. See the subversion repository: <http://anonsvn.jboss.org/repos/dna/trunk/extensions/dna-connector-inmemory/>

You can create your Maven project any way you'd like. For examples, see the [Maven 2 documentation](http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project) [http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-graph</artifactId>
  <version>0.3</version>
</dependency>
```

This is the only dependency required for compiling a connector - Maven pulls in all of the dependencies needed by the 'dna-graph' artifact. Of course, you'll still have to add dependencies for any library your connector needs to talk to its underlying system.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-graph</artifactId>
  <version>0.3</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-common</artifactId>
  <version>0.3</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
```



```

<version>1.1</version>
<scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.4.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
  <scope>test</scope>
</dependency>

```

Testing JBoss DNA connectors does not require a JCR repository or the JBoss DNA services. (For more detail, see the [testing section](#).) However, if you want to do integration testing with a JCR repository and the JBoss DNA services, you'll need additional dependencies (e.g., `dna-repository` and any other extensions).

At this point, your project should be set up correctly, and you're ready to move on to [writing the Java implementation](#) for your connector.

5.4.2. Implementing a `RepositorySource`

As mentioned earlier, a *connector* consists of the Java code that is used to access content from a system. Perhaps the most important class that makes up a connector is the implementation of the [RepositorySource](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html>]. This class is analogous to JDBC's `DataSource` in that it is instantiated to represent a single instance of a system that will be accessed, and it contains enough information (in the form of JavaBean properties) so that it can create connections to the source.

Why is the [RepositorySource](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html>] implementation a JavaBean? Well, this is the class that is instantiated, usually reflectively, and so a no-arg constructor is required. Using JavaBean properties makes it possible to reflect upon the object's class to determine the properties that can be set (using setters) and read (using getters). This means that an administrative application can instantiate, configure, and manage the objects that represent the actual sources, without having to know anything about the actual implementation.

So, your connector will need a public class that implements [RepositorySource](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/>

[apidocs/org/jboss/dna/graph/connectors/RepositorySource.html](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html)] and provides JavaBean properties for any kind of inputs or options required to establish a connection to and interact with the underlying source. Most of the semantics of the class are defined by the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] and inherited interface. However, there are a few characteristics that are worth mentioning here.

5.4.2.1. Cache policy

Each connector is responsible for determining whether and how long DNA is to cache the content made available by the connector. This is referred to as the *caching policy*, and consists of a *time to live* value representing the number of milliseconds that a piece of data may be cached. After the TTL has passed, the information is no longer used.

DNA allows a connector to use a flexible and powerful caching policy. First, each connection returns the *default* caching policy for all information returned by that connection. Often this policy can be configured via properties on the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation. This is optional, meaning the connector can return `null` if it does not wish to have a default caching policy.

Second, the connector is able to override its default caching policy on individual requests (which we'll cover in the [next section](#)). Again, this is optional, meaning that a null caching policy on a request implies that the request has no overridden caching policy.

Third, if the connector has no default caching policy and none is set on the individual requests, DNA uses whatever caching policy is set up for that component using the connector. For example, the federating connector allows a default caching policy to be specified, and this policy is used should the sources being federated not define their own caching policy.

In summary, a connector has total control over whether and for how long the information it provides is cached.

5.4.2.2. Leveraging JNDI

Sometimes it is necessary (or easier) for a [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation to look up an object in JNDI. One example of this is the JBoss Cache connector: while the connector can instantiate a new JBoss Cache instance, more interesting use cases involve JBoss Cache instances that are set up for clustering and replication, something that is generally difficult to configure in a single JavaBean. Therefore the [JBossCacheSource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/jboss/cache/JBossCacheSource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/connector/jboss/cache/JBossCacheSource.html] has optional JavaBean properties that define how it is to look up a JBoss Cache instance in JNDI.

This is a simple pattern that you may find useful in your connector. Basically, if your source implementation can look up an object in JNDI, simply use a single JavaBean String property that defines the full name that should be used to locate that object in JNDI. Usually it's best to include

"Jndi" in the JavaBean property name so that administrative users understand the purpose of the property. (And some may suggest that any optional property also use the word "optional" in the property name.)

5.4.2.3. Capabilities

Another characteristic of a [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation is that it provides some hint as to whether it supports several features. This is defined on the interface as a method that returns a [RepositorySourceCapabilities](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceCapabilities.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceCapabilities.html] object. This class currently provides methods that say whether the connector supports updates, whether it supports same-name-siblings (SNS), and whether the connector supports listeners and events.

Note that these may be hard-coded values, or the connector's response may be determined at runtime by various factors. For example, a connector may interrogate the underlying system to decide whether it can support updates.

The [RepositorySourceCapabilities](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceCapabilities.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceCapabilities.html] can be used as is (the class is immutable), or it can be subclassed to provide more complex behavior. It is important, however, that the capabilities remain constant throughout the lifetime of the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instance.



Note

Why a concrete class and not an interface? By using a concrete class, connectors inherit the default behavior. If additional capabilities need to be added to the class in future releases, connectors may not have to override the defaults. This provides some insulation against future enhancements to the connector framework.

5.4.2.4. Security and authentication

As we'll see in the next section, the main method connectors have to process requests takes an [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html], which contains the JAAS security information of the subject performing the request. This means that the connector can use this to determine authentication and authorization information for each request.

Sometimes that is not sufficient. For example, it may be that the connector needs its own authorization information so that it can establish a connection (even if user-level privileges still use the [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] provided with each request). In this case,

the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation will probably need JavaBean properties that represent the connector's authentication information. This may take the form of a username and password, or it may be properties that are used to delegate authentication to JAAS. Either way, just realize that it's perfectly acceptable for the connector to require its own security properties.

5.4.3. Implementing a `RepositoryConnection`

One job of the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] implementation is to create connections to the underlying sources. Connections are represented by classes that implement the [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] interface, and creating this class is the next step in writing a repository connector. This is what we'll cover in this section.

The [RepositoryConnection](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] interface is pretty straightforward:

```
/**
 * A connection to a repository source.
 *
 * <p>
 * These connections need not support concurrent operations by multiple threads.
 * </p>
 */
@NotThreadSafe
public interface RepositoryConnection [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html] {

    /**
     * Get the name for this repository source. This value should be the same as that returned
     * by the same RepositorySource [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] that created
     this connection.
     *
     * @return the identifier; never null or empty
     */
    String getSourceName();

    /**
```

```

    * Return the transactional resource associated with this connection. The transaction manager
    * will use this resource to manage the participation of this connection in a distributed transaction.
    *
    * @return the XA resource, or null if this connection is not aware of distributed transactions
    */
    XAResource getXAResource();

    /**
    * Ping the underlying system to determine if the connection is still valid and alive.
    *
    * @param time the length of time to wait before timing out
    * @param unit the time unit to use; may not be null
    * @return true if this connection is still valid and can still be used, or false otherwise
    * @throws InterruptedException if the thread has been interrupted during the operation
    */
    boolean ping( long time, TimeUnit unit ) throws InterruptedException;

    /**
    * Set the listener that is to receive notifications to changes to content within this source.
    *
    * @param listener the new listener, or null if no component is interested in the change
    notifications
    */
    void setListener( RepositorySourceListener listener );

    /**
    * Get the default cache policy for this repository. If none is provided, a global cache policy
    * will be used.
    *
    * @return the default cache policy
    */
    CachePolicy getDefaultCachePolicy();

    /**
    * Execute the supplied commands against this repository source.
    *
    * @param context the environment in which the commands are being executed; never null
    * @param request the request to be executed; never null
    * @throws RepositorySourceException if there is a problem loading the node data
    */

```

```
void execute( ExecutionContext context,
             Request request ) throws ;

/**
 * Close this connection to signal that it is no longer needed and that any accumulated
 * resources are to be released.
 */
void close();
}
```

While most of these methods are straightforward, a few warrant additional information. The `ping(...)` allows DNA to check the connection to see if it is alive. This method can be used in a variety of situations, ranging from verifying that a [`RepositorySource`](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html)'s JavaBean properties are correct to ensuring that a connection is still alive before returning the connection from a connection pool.

DNA hasn't yet defined the event mechanism, so connectors don't have any methods to invoke on the [`RepositorySourceListener`](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceListener.html). This will be defined in the next release, so feel free to manage the listeners now. Note that by default the [`RepositorySourceCapabilities`](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySourceCapabilities.html) returns `false` for `supportsEvents()`.

The most important method on this interface, though, is the `execute(...)` method, which serves as the mechanism by which the component using the connector access and manipulates the content exposed by the connector. The first parameter to this method is the [`ExecutionContext`](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html), which contains the information about environment as well as the subject performing the request. This was discussed [earlier](#).

The second parameter, however, represents a request that is to be processed by the connector. Request objects can take many different forms, as there are different classes for each kind of request (see the table below). Each request contains the information a connector needs to do the processing, and it also is the place where the connector places the results (or the error, if one occurs).

How do the requests reference a node (or nodes)? Since requests are coming from a client, the client may identify a particular node using a [`Location`](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html) object that is created with:

- the `Path` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html>] to the node; or
- one or more *identification properties* that are likely source-specific and that are represented with `Property` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html>] objects; or
- a combination of both.

So, when a client knows the path or the identification properties, they can create a `Location` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html>]. However, all of the requests return `Location` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html>] objects, so often times the client simply uses the location from a previous request. Since `Location` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html>] is an immutable class, it is perfectly safe to reuse them.

One more thing about locations: while the request may have an incomplete location (e.g., a path but no identification properties), the connector is expected to set on the request the *actual* location that contains the path and all identification properties. So as long as the client reuses the actual locations in subsequent requests, the connectors will have the benefit of having both the path and identification properties. Connectors can then be written to leverage this information, although the connector should still perform as expected when requests have incomplete locations.

Table 5.4. Types of Requests

Name	Description
<code>ReadNodeRequest</code>	A request to read from the source a node's properties and children. The node may be specified by path and/or by identification properties. The connector returns all properties and the locations for all children, or sets a <code>PathNotFoundException</code> [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node did not exist. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties).
<code>ReadAllPropertiesRequest</code>	A request to read from the source all of the properties of a node. The node may be specified by path and/or by identification properties. The connector returns all properties that were found on the node,

Name	Description
	<p>or sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node did not exist. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties).</p>
ReadPropertyRequest	<p>A request to read from the source a single property of a node. The node may be specified by path and/or by identification properties, and the property is specified by name. The connector returns the property if found on the node, or sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node or property did not exist. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties).</p>
ReadAllChildrenRequest	<p>A request to read from the source all of the children of a node. The node may be specified by path and/or by identification properties. The connector returns an ordered list of locations for each child found on the node, an empty list if the node had no children, or sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node did not exist. If the node is found, the connector sets on the request the actual location of the parent node (including the path and identification properties).</p>
ReadBlockOfChildrenRequest	<p>A request to read from the source a block of children of a node, starting with the n^{th}</p>

Name	Description
	<p>children. This is designed to allow paging through the children, which is much more efficient for large numbers of children. The node may be specified by path and/or by identification properties, and the block is defined by a starting index and a count (i.e., the block size). The connector returns an ordered list of locations for each of the node's children found in the block, or an empty list if there are no children in that range. The connector also sets on the request the actual location of the parent node (including the path and identification properties) or sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the parent node did not exist.</p>
ReadNextBlockOfChildrenRequest	<p>A request to read from the source a block of children of a node, starting with the children that immediately follow a previously-returned child. This is designed to allow paging through the children, which is much more efficient for large numbers of children. The node may be specified by path and/or by identification properties, and the block is defined by the location of the node immediately preceding the block and a count (i.e., the block size). The connector returns an ordered list of locations for each of the node's children found in the block, or an empty list if there are no children in that range. The connector also sets on the request the actual location of the parent node (including the path and identification properties) or sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the parent node did not exist.</p>

Name	Description
ReadBranchRequest	<p>A request to read a portion of a subgraph that has as its root a particular node, up to a maximum depth. This request is an efficient mechanism when a branch (or part of a branch) is to be navigated and processed, and replaces some non-trivial code to read the branch iteratively using multiple <code>ReadNodeRequest</code>s. The connector reads the branch to the specified maximum depth, returning the properties and children for all nodes found in the branch. The connector also sets on the request the actual location of the branch's root node (including the path and identification properties). The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node at the top of the branch does not exist.</p>
CreateNodeRequest	<p>A request to create a node at the specified location and setting on the new node the properties included in the request. The connector creates the node at the desired location, adjusting any same-name-sibling indexes as required. (If an SNS index is provided in the new node's location, existing children with the same name after that SNS index will have their SNS indexes adjusted. However, if the requested location does not include a SNS index, the new node is added after all existing children, and it's SNS index is set accordingly.) The connector also sets on the request the actual location of the new node (including the path and identification properties).. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/</p>

Name	Description
	<code>PathNotFoundException.html</code>] error on the request if the parent node does not exist.
<code>RemovePropertiesRequest</code>	A request to remove a set of properties on an existing node. The request contains the location of the node as well as the names of the properties to be removed. The connector performs these changes and sets on the request the actual location (including the path and identification properties) of the node. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node does not exist.
<code>UpdatePropertiesRequest</code>	A request to set or update properties on an existing node. The request contains the location of the node as well as the properties to be set and those to be deleted. The connector performs these changes and sets on the request the actual location (including the path and identification properties) of the node. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node does not exist.
<code>RenameNodeRequest</code>	A request to change the name of a node. The connector changes the node's name, adjusts all SNS indexes accordingly, and returns the actual locations (including the path and identification properties) of both the original location and the new location. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node does not exist.

Name	Description
CopyBranchRequest	<p>A request to copy a portion of a subgraph that has as its root a particular node, up to a maximum depth. The connector copies the branch from the original location, up to the specified maximum depth, and places a copy of the node as a child of the new location. The connector also sets on the request the actual location (including the path and identification properties) of the original location as well as the location of the new copy. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node at the top of the branch does not exist.</p>
MoveBranchRequest	<p>A request to move a subgraph that has a particular node as its root. The connector moves the branch from the original location and places it as child of the specified new location. The connector also sets on the request the actual location (including the path and identification properties) of the original and new locations. The connector will adjust SNS indexes accordingly. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node that is to be moved or the new location do not exist.</p>
DeleteBranchRequest	<p>A request to delete an entire branch specified by a single node's location. The connector deletes the specified node and all nodes below it, and sets the actual location, including the path and identification properties, of the node that was deleted. The connector sets a PathNotFoundException [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node does not exist.</p>

Name	Description
	apidocs/org/jboss/dna/graph/properties/PathNotFoundException.html] error on the request if the node being deleted does not exist.
CompositeRequest	A request that actually comprises multiple requests (none of which will be a composite). The connector simply processes all of the requests in the composite request, but should set on the composite request any error (usually the first error) that occurs during processing of the contained requests.

Although there are over a dozen different kinds of requests, we do anticipate adding more in future releases. For example, DNA will likely support searching repository content in sources through an additional subclass of `Request` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html>]. Getting the version history for a node will likely be another kind of request added in an upcoming release.

A connector is technically free to implement the `execute(...)` method in any way, as long as the semantics are maintained. But DNA provides a `RequestProcessor` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html>] class that can simplify writing your own connector and at the same time help insulate your connector from new kinds of requests that may be added in the future. The `RequestProcessor` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html>] is an abstract class that defines a `process(...)` method for each concrete `Request` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html>] subclass. In other words, there is a `process(CompositeRequest)` method, a `process(ReadNodeRequest)` method, and so on.

To use this in your connector, simply create a subclass of `RequestProcessor` [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html>], overriding all of the abstract methods and optionally overriding any of the other methods that have a default implementation.



Note

In many cases, the default implementations of the `process(...)` methods are *sufficient* but probably not efficient or optimum. If that is the case, simply provide your own methods that perform the request in a manner that is efficient for your source. However, if performance is not a big issue, all of the concrete methods

will provide the correct behavior. And remember, you can always provide better implementations later.

Then, in your connector's `execute(ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html], Request [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html])` method, instantiate your [RequestProcessor](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html] subclass and call its `process(Request [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html])` method, passing in the `execute(...)` method's [Request](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] parameter. The [RequestProcessor](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html] will determine the appropriate method given the actual [Request](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] object and will then invoke that method:

```
public void execute( final ExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] context,
                    final Request [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] request ) throws
    RepositorySourceException {
    RequestProcessor processor = new RequestProcessor(context);
    processor.process(request);
}
```

If you do this, the bulk of your connector implementation may be in the [RequestProcessor](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html] implementation methods. This not only is pretty maintainable, it also lends itself to easier testing. And should any new request types be added in the future, your connector may work just fine without any changes. In fact, if the [RequestProcessor](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/processor/RequestProcessor.html] class can implement meaningful methods for those new request types, your connector may "just work". Or, at least your connector will still be binary compatible, even if your connector won't support any of the new features.

Finally, how should the connector handle exceptions? As mentioned above, each [Request](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html] object has a slot where the connector can set any exception encountered during processing. This not only handles the exception, but in the case of [CompositeRequest](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/CompositeRequest.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/CompositeRequest.html]s it also correctly associates the problem with the request. However, it is perfectly acceptable to throw an exception if the connection becomes invalid (e.g., there is a communication failure) or if a fatal error would prevent subsequent requests from being processed.

5.4.4. Testing custom connectors

Testing connectors is not really that much different than testing other classes. Using mocks may help to isolate your instances so you can create more unit tests that don't require the underlying source system.

However, there may be times when you have to use the underlying source system in your tests. If this is the case, we recommend using Maven integration tests, which run at a different point in the Maven lifecycle. The benefit of using integration tests is that by convention they're able to rely upon external systems. Plus, your unit tests don't become polluted with slow-running tests that break if the external system is not available.

5.4.5. Configuring and deploying custom connectors

After building your connector project, you need to configure the JBoss DNA components your application is using so that they use your connector. In a lot of cases, this will entail instantiating your connector's [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] class, setting the various properties, and registering it with a [RepositoryLibrary](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html]. Or, it will entail using a configuration repository to use your source and letting [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html] instantiate and set up your [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html] instance. Or, you can just instantiate and set it up manually, passing the instance to whatever component needs it.

And of course you have to make the JAR file containing your connector (as well as any dependency JARs) available to your application's classpath.

5.5. Graph API for using connectors

So far we've talked about repositories, repository connectors, and how connectors respond to the different kinds of requests. Normally you'd code to the JCR API and use our JCR implementation. However, what does your code look like if you want to use the connectors directly, without using

our JCR implementation? After all, you may be a contributor to JBoss DNA, or you may want to take advantage of our connectors without all the overhead of JCR.

One option, of course, is to explicitly create the different requests and pass them to the connector's `execute(...)` method. While this is the most efficient approach (and one taken in some key DNA components), you probably want something that is much less verbose and much easier to use. This is where the DNA graph API comes in.

JBoss DNA's *Graph API* was designed as a lightweight public API for working with graph information, and it insulates components from the underlying requests and interacting with connectors. The *Graph* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html>] class is the primary class in API, and each instance represents a single, independent view of the graph of content from a single connector. *Graph* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html>] instances return snapshots of state, and those snapshots never change after they're retrieved. To obtain a *Graph* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html>] instance, use the static `create(...)` method, supplying the name of the source, a *RepositoryConnectionFactory* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html>] from which a *RepositoryConnection* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnection.html>] can be obtained, and the *ExecutionContext* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html>].

The *Graph* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html>] class basically represents an *internal domain specific language (DSL)* [<http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>], designed to be easy to use in an application. The Graph API makes extensive use of interfaces and method chaining, so that methods return a concise interface that has only those methods that make sense at that point. In fact, this should be really easy if your IDE has code completion. Just remember that under the covers, a *Graph* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html>] is just building *Request* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/requests/Requests.html>] objects, submitting them to the connector, and then exposing the results.

Let's look at some examples of how the Graph API works. This first example returns a map of properties (keyed by property name) for a node at a specific *Path* [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html>]:

```
Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] path = ...
```



```
Map<Name [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Name.html], Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html]>
propertiesByName = graph.getPropertiesByName().on(path);
```

This next example shows how the graph can be used to obtain and loop over the properties of a node:

```
Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] path = ...
for ( Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] property : graph.getProperties().on(path) ) {
    ...
}
```

Likewise, the next example shows how the graph can be used to obtain and loop over the children of a node:

```
Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] path = ...
for ( Location [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html] child : graph.getChildren().of(path) ) {
    Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] childPath = child.getPath();
    ...
}
```

Notice that the examples pass a [Path](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] instance to the `on(...)` and `of(...)` methods. Many of the Graph API methods take a variety of parameter types, including `String`, [Path](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html]s, [Location](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html]s, [UUID](http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html], or [Property](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] parameters. This should make it easy to use in many different situations.

Of course, changing content is more interesting and offers more interesting possibilities. Here are a few examples:

```
Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] path = ...
Location [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html] location = ...
Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] idProp1 = ...
Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] idProp2 = ...
UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid = ...
graph.move(path).into(idProp1, idProp2);
graph.copy(path).into(location);
graph.delete(uuid);
graph.delete(idProp1,idProp2);
```

The methods shown above work immediately, as soon as each request is built. However, there is another way to use the [Graph](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html) object, and that is in a *batch* mode. Simply create a [Graph.Batch](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.Batch.html) object using the `batch()` method, create the requests on that batch object, and then execute all of the commands on the batch by calling its `execute()` method. That `execute()` method returns a [Results](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Results.html) interface that can be used to read the node information retrieved by the batched requests.

Method chaining works really well with the batch mode, since multiple commands can be assembled together very easily:

```
Path [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Path.html] path = ...
String path2 = ...
Location [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html] location = ...
Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] idProp1 = ...
Property [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/properties/Property.html] idProp2 = ...
UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid = ...
graph.batch().move(path).into(idProp1,
idProp2).and().copy(path2).into(location).and().delete(uuid).execute();
Results [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Results.html] results = graph.batch().read(path2).
```

```
        and().readChildren().of(idProp1,idProp2).
        and().readSugraphOfDepth(3).at(uuid2).
        execute();
for ( Location [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Location.html] child : results.getNode(path2) ) {
    ...
}
```

Of course, this section provided just a hint of the Graph API. The [Graph](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html] interface is actually quite complete and offers a full-featured approach for reading and updating a graph. For more information, see the [Graph](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/Graph.html] JavaDocs.

5.6. Summary

In this chapter, we covered all the aspects of JBoss DNA repositories, including the connector framework, how DNA's JCR implementation works with connectors, what connectors are available (and how to use them), and how to write your own connector. So now that you know how to set up and use JBoss DNA repositories, the [next chapter](#) describes how you can leverage JBoss DNA's JCR implementation.

Content Repositories for Java (JCR)

The [Content Repository for Java technology API](http://www.jcp.org/en/jsr/detail?id=170) [http://www.jcp.org/en/jsr/detail?id=170] provides a standard Java API for working with content repositories. Abbreviated "JCR", this API was developed as part of the Java Community Process under [JSR-170](http://www.jcp.org/en/jsr/detail?id=170) [http://www.jcp.org/en/jsr/detail?id=170] (JCR 1.0) and is being revised under [JSR-283](http://www.jcp.org/en/jsr/detail?id=283) [http://www.jcp.org/en/jsr/detail?id=283]. JBoss DNA provides a JCR 1.0 implementation that allows you to work with the contents of a repository using the JCR API. For information about how to use the JCR API, please see the [JSR-170](http://www.jcp.org/en/jsr/detail?id=170) [http://www.jcp.org/en/jsr/detail?id=170] specification.

6.1. Obtaining JCR repositories

The JCR API doesn't define how your application first obtains a reference to a Repository implementation. With JBoss DNA, you simply create a [JcrRepository](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html] object and supply an [ExecutionContextFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html] and a [RepositoryConnectionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html] (such as a [RepositoryLibrary](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryLibrary.html] or [RepositoryService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html]). Since [JcrRepository](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html] implements the JCR Repository interface, from this point forward you can just use the standard JCR API.



Note

For more information about the [ExecutionContextFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContextFactory.html] and [RepositoryConnectionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositoryConnectionFactory.html] classes, see the chapter on [setting up a JBoss DNA environment](#) and [setting up the RepositoryService](#) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/RepositoryService.html].

6.2. Creating JCR sessions

Creating sessions is done using a Repository one of its `login(...)` methods, where the name of the workspace corresponds to the name of

the [RepositorySource](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/connectors/RepositorySource.html]:

```
JcrRepository [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html] jcrRepository = new JcrRepository [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/JcrRepository.html](contextFactory, sources);
Session session = jcrRepository.login(sourceName);
```

Now, this code doesn't do any authentication; it essentially trusts the caller has the appropriate privileges. Normally, your application will need to authenticate the user, so let's look at how that's done.

As we mentioned in the [security section](#), JBoss DNA uses JAAS for authentication and authorization. So how does this work with the JCR API?

The JCR API defines a Credentials marker interface, an instance of which can be passed to the `Session.login(...)` method. Rather than provide a concrete implementation of this interface, JBoss DNA allows you to pass any implementation of Credentials that also has one of the following methods:

- `getLoginContext()` that returns a [LoginContext](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] instance.
- `getAccessControlContext()` that returns a [AccessControlContext](http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html] instance.

This way, your application can obtain the JAAS [LoginContext](http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] or [AccessControlContext](http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html] however it wants, and then merely passes that into DNA through the JCR Credentials. No interfaces or classes specific to JBoss DNA are required.

The following code shows how this is done, using an anonymous inner class for the Credentials implementation.

```
CallbackHandler [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/callback/CallbackHandler.html] callbackHandler = // as needed by your app, according to JAAS
final LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] loginContext = new LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html]("MyAppContextName",callbackHandler);

// Now pass to JBoss DNA to create a JCR Session ...
```

```
Credentials credentials = new Credentials() {  
    public LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/  
LoginContext.html] getLoginContext() { return loginContext; }  
};  
JcrRepository [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/  
apidocs/org/jboss/dna/jcr/JcrRepository.html] jcrRepository = new JcrRepository [http://  
www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/jcr/  
JcrRepository.html](contextFactory, sources);  
Session session = jcrRepository.login(credentials, sourceName);
```

6.3. Summary

In this chapter, we covered how to use JCR with JBoss DNA. Now that you know how JBoss DNA repositories work, and how to use JCR to work with DNA repositories, we'll move on in the [next chapter](#) to describing in detail how the sequencing of file content works.

Sequencing content

As we've mentioned before, JBoss DNA is able to work with existing JCR repositories. Your client applications make changes to the information in those repositories, and JBoss DNA automatically uses its sequencers to extract additional information from the uploaded files.

This chapter discusses the sequencing features of JBoss DNA and the components that are involved.

7.1. Sequencing Service

The JBoss DNA *sequencing service* is the component that manages the *sequencers*, reacting to changes in JCR repositories and then running the appropriate sequencers. This involves processing the changes on a node, determining which (if any) sequencers should be run on that node, and for each sequencer constructing the execution environment, calling the sequencer, and saving the information generated by the sequencer.



Note

Configuring JBoss DNA services is a bit more manual than is ideal. As you'll see, JBoss DNA uses dependency injection to allow a great deal of flexibility in how it can be configured and customized. But this flexibility makes it more difficult for you to use. We understand this, and will soon provide a much easier way to set up and manage JBoss DNA. Current plans are to use the *JBoss Microcontainer* [<http://www.jboss.org/jbossmc>] along with a configuration repository.

To set up the sequencing service, an instance is created, and dependent components are injected into the object. This includes among other things:

- An *execution context* that defines the context in which the service runs, including a factory for JCR sessions given names of the repository and workspace. This factory must be configured, and is how JBoss DNA knows about your JCR repositories and how to connect to them. More on this a bit later.
- An optional *factory for class loaders* used to load sequencers. If no factory is supplied, the service uses the current thread's context class loader (or if that is null, the class loader that loaded the sequencing service class).
- An *ExecutorService* [<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ExecutorService.html>] used to execute the sequencing activities. If none is supplied, a new single-threaded executor is created by calling `Executors.newSingleThreadExecutor()`. (This can easily be changed by subclassing and overriding the `SequencerService.createDefaultExecutorService()` method.)

- Filters for sequencers and events. By default, all sequencers are considered for "node added", "property added" and "property changed" events.

As mentioned above, the [ExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/ExecutionContext.html] provides access to a [SessionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SessionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SessionFactory.html] that is used by JBoss DNA to establish sessions to your JCR repositories. Two implementations are available:

- The [JndiSessionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/JndiSessionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/JndiSessionFactory.html] looks up JCR Repository instances in JNDI using names that are supplied when creating sessions. This implementation also has methods to set the JCR Credentials for a given workspace name.
- The [SimpleSessionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SimpleSessionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SimpleSessionFactory.html] has methods to register the JCR Repository instances with names, as well as methods to set the JCR Credentials for a given workspace name.

You can use the [BasicJcrExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/BasicJcrExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/BasicJcrExecutionContext.html] implementation of [JcrExecutionContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/JcrExecutionContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/JcrExecutionContext.html] and supply a [SessionFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SessionFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SessionFactory.html] instance, or you can provide your own implementation.

Here's an example of how to instantiate and configure the [SequencingService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html]:

```
SimpleSessionFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SimpleSessionFactory.html] sessionFactory = new
SimpleSessionFactory [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/SimpleSessionFactory.html]();
sessionFactory.registerRepository("Main Repository", this.repository);
Credentials credentials = new ("jsmith", "secret".toCharArray());
sessionFactory.registerCredentials("Main Repository/Workspace1", credentials);
ExecutionContext executionContext = new BasicJcrExecutionContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/util/BasicJcrExecutionContext.html](sessionFactory);

// Create the sequencing service, passing in the execution context ...
SequencingService [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html] sequencingService =
```

```
new SequencingService [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html]());
sequencingService.setExecutionContext(executionContext);
```

After the sequencing service is created and configured, it must be started. The [SequencingService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html] has an *administration object* (that is an instance of [ServiceAdministrator](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html]) with `start()`, `pause()`, and `shutdown()` methods. The latter method will close the queue for sequencing, but will allow sequencing operations already running to complete normally. To wait until all sequencing operations have completed, simply call the `awaitTermination` method and pass it the maximum amount of time you want to wait.

```
sequencingService.getAdministrator().start();
```

The JBoss DNA services are utilizing resources and threads that must be released before your application is ready to shut down. The safe way to do this is to simply obtain the [ServiceAdministrator](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/services/ServiceAdministrator.html] for each service (via the `getServiceAdministrator()` method) and call `shutdown()`. As previously mentioned, the shutdown method will simply prevent new work from being processed and will not wait for existing work to be completed. If you want to wait until the service completes all its work, you must wait until the service terminates. Here's an example that shows how this is done:

```
// Shut down the service and wait until it's all shut down ...
sequencingService.getAdministrator().shutdown();
sequencingService.getAdministrator().awaitTermination(5, TimeUnit.SECONDS);

// Shut down the observation service ...
observationService.getAdministrator().shutdown();
observationService.getAdministrator().awaitTermination(5, TimeUnit.SECONDS);
```

7.2. Sequencer Configurations

The sequencing service must also be configured with the sequencers that it will use. This is done using the `addSequencer(SequencerConfig)` method and passing a [SequencerConfig](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] instance that you create.

Here's the code that defines 3 sequencer configurations: 1 that places image metadata into `/images/<filename>`, another that places MP3 metadata into `/mp3s/<filename>`, and a third that places a structure that represents the classes, methods, and attributes found within Java source into `/java/<filename>`.

```
String name = "Image Sequencer";
String desc = "Sequences image files to extract the characteristics of the image";
String classname = "org.jboss.dna.sequencer.images.ImageMetadataSequencer";
String[] classpath = null; // Use the current classpath
String[] pathExpressions = {"/*.(jpg|jpeg|gif|bmp|pcx|png)[*]"/jcr:content[@jcr:data] => /images/
$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] imageSequencerConfig
    = new SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/
docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc,
    classname,
                                classpath, pathExpressions);
sequencingService.addSequencer(imageSequencerConfig);

name = "MP3 Sequencer";
desc = "Sequences MP3 files to extract the ID3 tags from the audio file";
classname = "org.jboss.dna.sequencer.mp3.Mp3MetadataSequencer";
pathExpressions = {"/*(.mp3[*]"/jcr:content[@jcr:data] => /mp3s/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] mp3SequencerConfig =
    new SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/
docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc,
    classname,
                                classpath, pathExpressions);
sequencingService.addSequencer(mp3SequencerConfig);

name = "Java Sequencer";
desc = "Sequences java files to extract the characteristics of the Java source";
classname = "org.jboss.dna.sequencer.java.JavaMetadataSequencer";
pathExpressions = {"/*(.java[*]"/jcr:content[@jcr:data] => /java/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] javaSequencerConfig =
    new SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/
docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc,
    classname,
                                classpath, pathExpressions);
this.sequencingService.addSequencer(javaSequencerConfig);
```

Each configuration defines several things, including the name, description, and sequencer implementation class. The configuration also defines the classpath information, which can be passed to the [ClassLoaderFactory](http://www.jboss.org/file-access/default/members/dna/freezezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html] to get a Java [ClassLoader](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html] with which the sequencer class can be loaded. (If no classpath information is provided, as is done in the code above, the application class loader is used.) The configuration also specifies the path expressions that identify the nodes that should be sequenced with the sequencer and where to store the output generated by the sequencer. Path expressions are pretty straightforward but are quite powerful, so before we go any further with the example, let's dive into path expressions in more detail.

7.2.1. Path Expressions

Path expressions consist of two parts: a selection criteria (or an input path) and an output path:

```
inputPath => outputPath
```

The *inputPath* part defines an expression for the path of a node that is to be sequenced. Input paths consist of '/' separated segments, where each segment represents a pattern for a single node's name (including the same-name-sibling indexes) and '@' signifies a property name.

Let's first look at some simple examples:

Table 7.1. Simple Input Path Examples

Input Path	Description
/a/b	Match node "b" that is a child of the top level node "a". Neither node may have any same-name-siblings.
/a/*	Match any child node of the top level node "a".
/a/*.txt	Match any child node of the top level node "a" that also has a name ending in ".txt".
/a/*	Match any child node of the top level node "a" that also has a name ending in ".txt".
/a/b@c	Match the property "c" of node "/a/b".
/a/b[2]	The second child named "b" below the top level node "a".
/a/b[2,3,4]	The second, third or fourth child named "b" below the top level node "a".
/a/b[*]	

Input Path	Description
	Any (and every) child named "b" below the top level node "a".
//a/b	Any node named "b" that exists below a node named "a", regardless of where node "a" occurs. Again, neither node may have any same-name-siblings.

With these simple examples, you can probably discern the most important rules. First, the '*' is a wildcard character that matches any character or sequence of characters in a node's name (or index if appearing in between square brackets), and can be used in conjunction with other characters (e.g., "*.txt").

Second, square brackets (i.e., '[' and ']') are used to match a node's same-name-sibling index. You can put a single non-negative number or a comma-separated list of non-negative numbers. Use '0' to match a node that has no same-name-siblings, or any positive number to match the specific same-name-sibling.

Third, combining two delimiters (e.g., "/"") matches any sequence of nodes, regardless of what their names are or how many nodes. Often used with other patterns to identify nodes at any level matching other patterns. Three or more sequential slash characters are treated as two.

Many input paths can be created using just these simple rules. However, input paths can be more complicated. Here are some more examples:

Table 7.2. More Complex Input Path Examples

Input Path	Description
/a/(b c d)	Match children of the top level node "a" that are named "a", "b" or "c". None of the nodes may have same-name-sibling indexes.
/a/b[c/d]	Match node "b" child of the top level node "a", when node "b" has a child named "c", and "c" has a child named "d". Node "b" is the selected node, while nodes "b" and "b" are used as criteria but are not selected.
/a/(b c d e)[f/g/@something]	Match node "/a/b/e", "/a/c/e", "/a/d/e", or "/a/e" when they also have a child "f" that itself has a child "g" with property "something". None of the nodes may have same-name-sibling indexes.

These examples show a few more advanced rules. Parentheses (i.e., '(' and ')') can be used to define a set of options for names, as shown in the first and third rules. Whatever part of the selected node's path appears between the parentheses is captured for use within the output path.

Thus, the first input path in the previous table would match node `/a/b`, and `"b"` would be captured and could be used within the output path using `"$1"`, where the number used in the output path identifies the parentheses.

Square brackets can also be used to specify criteria on a node's properties or children. Whatever appears in between the square brackets does not appear in the selected node.

Let's go back to the previous code fragment and look at the first path expression:

```
//(*.(jpg|jpeg|gif|bmp|pcx|png)[*])/jcr:content[@jcr:data] => /images/$1
```

This matches a node named `"jcr:content"` with property `"jcr:data"` but no siblings with the same name, and that is a child of a node whose name ends with `".jpg"`, `".jpeg"`, `".gif"`, `".bmp"`, `".pcx"`, or `".png"` that may have any same-name-sibling index. These nodes can appear at any level in the repository. Note how the input path capture the filename (the segment containing the file extension), including any same-name-sibling index. This filename is then used in the output path, which is where the sequenced content is placed.

7.3. JBoss DNA Sequencers

JBoss DNA includes a number of sequencers "out of the box". These sequencers can be used within your application to sequence a variety of common file formats. To use them, the only thing you have to do is define the appropriate sequencer configurations and include the appropriate JAR files.

7.3.1. Image sequencer

A sequencer that extracts metadata from JPEG, GIF, BMP, PCX, PNG, IFF, RAS, PBM, PGM, PPM and PSD image files. This sequencer extracts the file format, image resolution, number of bits per pixel and optionally number of images, comments and physical resolution, and then writes this information into the repository using the following structure:

- **image:metadata** node of type `image:metadata`
- **jcr:mimeType** - optional string property for the mime type of the image
 - **jcr:encoding** - optional string property for the encoding of the image
- **image:formatName** - string property for the name of the format
- **image:width** - optional integer property for the image's width in pixels
- **image:height** - optional integer property for the image's height in pixels
- **image:bitsPerPixel** - optional integer property for the number of bits per pixel
- **image:progressive** - optional boolean property specifying whether the image is stored in a progressive (i.e., interlaced) form

- **image:numberOfImages** - optional integer property for the number of images stored in the file; defaults to 1
- **image:physicalWidthDpi** - optional integer property for the physical width of the image in dots per inch
- **image:physicalHeightDpi** - optional integer property for the physical height of the image in dots per inch
- **image:physicalWidthInches** - optional double property for the physical width of the image in inches
- **image:physicalHeightInches** - optional double property for the physical height of the image in inches

This structure could be extended in the future to add EXIF and IPTC metadata as child nodes. For example, EXIF metadata is structured as tags in directories, where the directories form something like namespaces, and which are used by different camera vendors to store custom metadata. This structure could be mapped with each directory (e.g. "EXIF" or "Nikon Makernote" or "IPTC") as the name of a child node, with the EXIF tags values stored as either properties or child nodes.

To use this sequencer, simply include the `dna-sequencer-images` JAR in your application and configure the Sequencing Service to use this sequencer using something similar to:

```
String name = "Image Sequencer";
String desc = "Sequences image files to extract the characteristics of the image";
String classname = "org.jboss.dna.sequencer.images.ImageMetadataSequencer";
String[] classpath = null; // Use the current classpath
String[] pathExpressions = {"/*.(jpg|jpeg|gif|bmp|pcx|png|iff|ras|pbm|pgm|ppm|psd)[*]"/
jcr:content[@jcr:data] => /images/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] sequencerConfig = new
    SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc, classname,
        classpath, pathExpressions);
sequencingService.addSequencer(sequencerConfig);
```

7.3.2. Microsoft Office document sequencer

This sequencer is included in JBoss DNA and processes Microsoft Office documents, including Excel spreadsheets and PowerPoint presentations. With presentations, the sequencer extracts the slides, titles, text and slide thumbnails. With spreadsheets, the sequencer extracts the names of the sheets. And, the sequencer extracts for all the files the general file information, including the name of the author, title, keywords, subject, comments, and various dates.



Note

Currently, Word documents are not supported. For more information and the latest status, see [DNA-153](http://jira.jboss.org/jira/browse/DNA-153) [http://jira.jboss.org/jira/browse/DNA-153].

To use this sequencer, simply include the `dna-sequencer-msoffice` JAR and all of the [POI](http://poi.apache.org/) [http://poi.apache.org/] JARs in your application and configure the Sequencing Service to use this sequencer using something similar to:

```
String name = "Microsoft Office Document Sequencer";
String desc = "Sequences MS Office documents, including spreadsheets and presentations";
String classname = "org.jboss.dna.sequencer.msoffice.MSOfficeMetadataSequencer";
String[] classpath = null; // Use the current classpath
String[] pathExpressions = {"/*.(doc|docx|ppt|pps|xls|*)/jcr:content[@jcr:data] => /msoffice/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] sequencerConfig = new
    SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc, classname,
        classpath, pathExpressions);
sequencingService.addSequencer(sequencerConfig);
```

7.3.3. ZIP archive sequencer

The ZIP file sequencer is included in JBoss DNA and extracts the files and folders contained in the ZIP archive file, extracting the files and folders into the repository using JCR's `nt:file` and `nt:folder` node types.

To use this sequencer, simply include the `dna-sequencer-zip` JAR in your application and configure the Sequencing Service to use this sequencer using something similar to:

```
String name = "ZIP Sequencer";
String desc = "Sequences ZIP archives to extract the files and folders";
String classname = "org.jboss.dna.sequencer.zip.ZipSequencer";
String[] pathExpressions = {"/*(.zip|*)/jcr:content[@jcr:data] => /zips/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] sequencerConfig = new
    SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc, classname,
        classpath, pathExpressions);
```

```
this.sequencingService.addSequencer(sequencerConfig);
```

7.3.4. Java source sequencer

One of the sequencers that included in JBoss DNA is the **dna-sequencer-java** subproject. This sequencer parses Java source code added to the repository and extracts the basic structure of the classes and enumerations defined in the code. This structure includes: the package structures, class declarations, class and member attribute declarations, class and member method declarations with signature (but not implementation logic), enumerations with each enumeration literal value, annotations, and JavaDoc information for all of the above. After extracting this information from the source code, the sequencer then writes this structure into the repository, where it can be further processed, analyzed, searched, navigated, or referenced.

To use this sequencer, simply include the `dna-sequencer-java` JAR (plus all of the JARs that it is dependent upon) in your application and configure the Sequencing Service to use this sequencer using something similar to:

```
String name = "Java Sequencer";
String desc = "Sequences java files to extract the characteristics of the Java source";
String classname = "org.jboss.dna.sequencer.java.JavaMetadataSequencer";
String[] classpath = null; // Use the current classpath
String[] pathExpressions = {"/*(*.java[*])/jcr:content[@jcr:data] => /java/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] sequencerConfig = new
    SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc, classname,
        classpath, pathExpressions);
this.sequencingService.addSequencer(sequencerConfig);
```

7.3.5. MP3 audio file sequencer

Another sequencer that is included in JBoss DNA is the **dna-sequencer-mp3** sequencer project. This sequencer processes MP3 audio files added to a repository and extracts the [ID3](http://www.id3.org/) metadata for the file, including the track's title, author, album name, year, and comment. After extracting this information from the audio files, the sequencer then writes this structure into the repository, where it can be further processed, analyzed, searched, navigated, or referenced.

To use this sequencer, simply include the `dna-sequencer-mp3` JAR and the [JAudioTagger](http://www.jthink.net/jaudiotagger/) library in your application and configure the Sequencing Service to use this sequencer using something similar to:

```
String name = "MP3 Sequencer";
String desc = "Sequences MP3 files to extract the ID3 tags of the audio file";
String classname = "org.jboss.dna.sequencer.mp3.Mp3MetadataSequencer";
String[] pathExpressions = {"/*(.mp3[*])/jcr:content[@jcr:data] => /mp3s/$1"};
SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html] sequencerConfig = new
    SequencerConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html](name, desc, classname,
        classpath, pathExpressions);
this.sequencingService.addSequencer(sequencerConfig);
```

7.3.6. JCR Compact Node Definition (CND) file sequencer

This sequencer is incomplete and is not currently usable. The purpose is to sequence JCR Compact Node Definition (CND) files to extract the node definitions with their property definitions, and inserting these into the repository using JCR standard notation.

7.4. Creating custom sequencers

The current release of JBoss DNA comes with six sequencers. However, it's very easy to create your own sequencers and to then configure JBoss DNA to use them in your own application.

Creating a custom sequencer involves the following steps:

1. Create a Maven 2 project for your sequencer;
2. Implement the [StreamSequencer](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] interface with your own implementation, and create unit tests to verify the functionality and expected behavior;
3. Add the sequencer configuration to the JBoss DNA [SequencingService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html] in your application as described in the [previous chapter](#), and
4. Deploy the JAR file with your implementation (as well as any dependencies), and make them available to JBoss DNA in your application.

It's that simple.

7.4.1. Creating the Maven 2 project

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already [set up to use Maven](#),

using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.



Note

JBoss DNA may provide in the future a Maven archetype for creating sequencer projects. If you'd find this useful and would like to help create it, please [join the community](#).

In lieu of a Maven archetype, you may find it easier to start with a small existing sequencer project. The **dna-sequencer-images** project is a small, self-contained sequencer implementation that has only the minimal dependencies. See the subversion repository: <http://anonsvn.jboss.org/repos/dna/trunk/extensions/dna-sequencer-images/>

You can create your Maven project any way you'd like. For examples, see the [Maven 2 documentation](#) [http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-common</artifactId>
  <version>0.3</version>
</dependency>
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-graph</artifactId>
  <version>0.3</version>
</dependency>
```

These are minimum dependencies required for compiling a sequencer. Of course, you'll have to add other dependencies that your sequencer needs.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<dependency>
  <groupId>org.jboss.dna</groupId>
```

```
<artifactId>dna-graph</artifactId>
<version>0.3</version>
<type>test-jar</type>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-common</artifactId>
  <version>0.3</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.4.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
  <scope>test</scope>
</dependency>
```

Testing JBoss DNA sequencers does not require a JCR repository or the JBoss DNA services. (For more detail, see the [testing section](#).) However, if you want to do integration testing with a JCR repository and the JBoss DNA services, you'll need additional dependencies for these libraries.

```
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-repository</artifactId>
  <version>0.1</version>
  <scope>test</scope>
</dependency>
<!-- Java Content Repository API -->
<dependency>
  <groupId>javax.jcr</groupId>
  <artifactId>jcr</artifactId>
  <version>1.0.1</version>
  <scope>test</scope>
</dependency>
<!-- Apache Jackrabbit (JCR Implementation) -->
<dependency>
  <groupId>org.apache.jackrabbit</groupId>
  <artifactId>jackrabbit-api</artifactId>
  <version>1.3.3</version>
  <scope>test</scope>
  <!-- Exclude these since they are included in JDK 1.5 -->
  <exclusions>
    <exclusion>
      <groupId>xml-apis</groupId>
      <artifactId>xml-apis</artifactId>
    </exclusion>
    <exclusion>
      <groupId>xerces</groupId>
      <artifactId>xercesImpl</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.jackrabbit</groupId>
  <artifactId>jackrabbit-core</artifactId>
  <version>1.3.3</version>
  <scope>test</scope>
  <!-- Exclude these since they are included in JDK 1.5 -->
  <exclusions>
    <exclusion>
      <groupId>xml-apis</groupId>
      <artifactId>xml-apis</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```

<exclusion>
  <groupId>xerces</groupId>
  <artifactId>xercesImpl</artifactId>
</exclusion>
</exclusions>
</dependency>

```

At this point, your project should be set up correctly, and you're ready to move on to [writing the Java implementation](#) for your sequencer.

7.4.2. Implementing the StreamSequencer interface

After creating the project and setting up the dependencies, the next step is to create a Java class that implements the [StreamSequencer](#) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] interface. This interface is very straightforward and involves a single method:

```

public interface StreamSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] {

    /**
     * Sequence the data found in the supplied stream, placing the output
     * information into the supplied map.
     *
     * @param stream the stream with the data to be sequenced; never null
     * @param output the output from the sequencing operation; never null
     * @param context the context for the sequencing operation; never null
     */
    void sequence( InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream, SequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerOutput.html] output,
        SequencerContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerContext.html] context );
}

```

The job of a stream sequencer is to process the data in the supplied stream, and place into the [SequencerOutput](#) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerOutput.html] any information that is to go into the JCR repository. JBoss DNA figures out when your sequencer should be called (of course, using the sequencing configuration you'll add in a bit), and then makes sure the generated information is saved in the correct place in the repository.

The [SequencerContext](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerContext.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerContext.html] provides information about the current sequencing operation, including the location and properties of the node being sequenced, the MIME type of the node being sequenced, and a location to record problems that aren't severe enough to warrant throwing an exception.

The [SequencerOutput](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerOutput.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerOutput.html] class is fairly easy to use. There are basically two methods you need to call. One method sets the property values, while the other sets references to other nodes in the repository. Use these methods to describe the properties of the nodes you want to create, using relative paths for the nodes and valid JCR property names for properties and references. JBoss DNA will ensure that nodes are created or updated whenever they're needed.

```
public interface SequencerOutput {
```

```
    /**
     * Set the supplied property on the supplied node. The allowable
     * values are any of the following:
     * - primitives (which will be autoboxed)
     * - String instances
     * - String arrays
     * - byte arrays
     * - InputStream instances
     * - Calendar instances
     *
     * @param nodePath the path to the node containing the property;
     * may not be null
     * @param property the name of the property to be set
     * @param values the value(s) for the property; may be empty if
     * any existing property is to be removed
     */
    void setProperty( String nodePath, String property, Object... values );

    /**
     * Set the supplied reference on the supplied node.
     *
     * @param nodePath the path to the node containing the property;
     * may not be null
     * @param property the name of the property to be set
     * @param paths the paths to the referenced property, which may be
     * absolute paths or relative to the sequencer output node;
```



```

* may be empty if any existing property is to be removed
*/
void setReference( String nodePath, String property, String... paths );
}

```

JBoss DNA will create nodes of type `nt:unstructured` unless you specify the value for the `jcr:primaryType` property. You can also specify the values for the `jcr:mixinTypes` property if you want to add mixins to any node.

For a complete example of a sequencer, let's look at the [ImageMetadataSequencer](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadataSequencer.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadataSequencer.html] implementation:

```

public class ImageMetadataSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadataSequencer.html]
    implements StreamSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] {

    public static final String METADATA_NODE = "image:metadata";
    public static final String IMAGE_PRIMARY_TYPE = "jcr:primaryType";
    public static final String IMAGE_MIXINS = "jcr:mixinTypes";
    public static final String IMAGE_MIME_TYPE = "jcr:mimeType";
    public static final String IMAGE_ENCODING = "jcr:encoding";
    public static final String IMAGE_FORMAT_NAME = "image:formatName";
    public static final String IMAGE_WIDTH = "image:width";
    public static final String IMAGE_HEIGHT = "image:height";
    public static final String IMAGE_BITS_PER_PIXEL = "image:bitsPerPixel";
    public static final String IMAGE_PROGRESSIVE = "image:progressive";
    public static final String IMAGE_NUMBER_OF_IMAGES = "image:numberOfImages";
    public static final String IMAGE_PHYSICAL_WIDTH_DPI = "image:physicalWidthDpi";
    public static final String IMAGE_PHYSICAL_HEIGHT_DPI = "image:physicalHeightDpi";
    public static final String IMAGE_PHYSICAL_WIDTH_INCHES = "image:physicalWidthInches";
        public static final String IMAGE_PHYSICAL_HEIGHT_INCHES =
            "image:physicalHeightInches";

    /**
     * {@inheritDoc}
     */

    public void sequence( InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream,
        SequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerOutput.html] output,

```

```
SequencerContext [http://www.jboss.org/file-access/default/members/dna/
freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/SequencerContext.html] context ) {
    ImageMetadata [http://www.jboss.org/file-access/default/members/dna/
freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadata.html] metadata =
    new ImageMetadata [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/
apidocs/org/jboss/dna/sequencer/image/ImageMetadata.html]();
    metadata.setInput(stream);
    metadata.setDetermineImageNumber(true);
    metadata.setCollectComments(true);

    // Process the image stream and extract the metadata ...
    if (!metadata.check()) {
        metadata = null;
    }
    // Generate the output graph if we found useful metadata ...
    if (metadata != null) {
        // Place the image metadata into the output map ...
        output.setProperty(METADATA_NODE, IMAGE_PRIMARY_TYPE, "image:metadata");
        // output.psetProperty(METADATA_NODE, IMAGE_MIXINS, "");
        output.setProperty(METADATA_NODE, IMAGE_MIME_TYPE, metadata.getMimeType());
        // output.setProperty(METADATA_NODE, IMAGE_ENCODING, "");
        output.setProperty(METADATA_NODE, IMAGE_FORMAT_NAME,
metadata.getFormatName());
        output.setProperty(METADATA_NODE, IMAGE_WIDTH, metadata.getWidth());
        output.setProperty(METADATA_NODE, IMAGE_HEIGHT, metadata.getHeight());
        output.setProperty(METADATA_NODE, IMAGE_BITS_PER_PIXEL,
metadata.getBitsPerPixel());
        output.setProperty(METADATA_NODE, IMAGE_PROGRESSIVE,
metadata.isProgressive());
        output.setProperty(METADATA_NODE, IMAGE_NUMBER_OF_IMAGES,
metadata.getNumberOfImages());
        output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_WIDTH_DPI,
metadata.getPhysicalWidthDpi());
        output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_HEIGHT_DPI,
metadata.getPhysicalHeightDpi());
        output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_WIDTH_INCHES,
metadata.getPhysicalWidthInch());
        output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_HEIGHT_INCHES,
metadata.getPhysicalHeightInch());
    }
}
}
```

Notice how the image metadata is extracted and the output graph is generated. A single node is created with the name `image:metadata` and with the `image:metadata` node type. No mixins are defined for the node, but several properties are set on the node using the values obtained from the image metadata. After this method returns, the constructed graph will be saved to the repository in all of the places defined by its configuration. (This is why only relative paths are used in the sequencer.)

7.4.3. Testing custom sequencers

The sequencing framework was designed to make testing sequencers much easier. In particular, the [StreamSequencer](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] interface does not make use of the JCR API. So instead of requiring a fully-configured JCR repository and JBoss DNA system, unit tests for a sequencer can focus on testing that the content is processed correctly and the desired output graph is generated.



Note

For a complete example of a sequencer unit test, see the `ImageMetadataSequencerTest` unit test in the `org.jboss.dna.sequencer.images` package of the `dna-sequencers-image` project.

The following code fragment shows one way of testing a sequencer, using JUnit 4.4 assertions and some of the classes made available by JBoss DNA. Of course, this example code does not do any error handling and does not make all the assertions a real test would.

```
StreamSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/StreamSequencer.html] sequencer = new
    ImageMetadataSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadataSequencer.html]();
MockSequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerOutput.html] output = new
    MockSequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerOutput.html]();
MockSequencerContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerContext.html] context = new
    MockSequencerContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerContext.html]();
InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream = null;
try {
    stream = this.getClass().getClassLoader().getResource("caution.gif").openStream();
    sequencer.sequence(stream,output,context); // writes to 'output'
}
```

```
assertThat(output.getPropertyValues("image:metadata", "jcr:primaryType"),
    is(new Object[] { "image:metadata"}));
assertThat(output.getPropertyValues("image:metadata", "jcr:mimeType"),
    is(new Object[] { "image/gif"}));
// ... make more assertions here
assertThat(output.hasReferences(), is(false));
} finally {
    stream.close();
}
```

It's also useful to test that a sequencer produces no output for something it should not understand:

```
sequencer = new ImageMetadataSequencer [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/sequencer/image/ImageMetadataSequencer.html]();
MockSequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerOutput.html] output = new
    MockSequencerOutput [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerOutput.html]();
MockSequencerContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerContext.html] context = new
    MockSequencerContext [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/sequencers/MockSequencerContext.html]();
InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream = null;
try {
    stream = this.getClass().getClassLoader().getResource("caution.pict").openStream();
    sequencer.sequence(stream,output,context); // writes to 'output'
    assertThat(output.getProperties(), is(false));
    assertThat(output.hasReferences(), is(false));
} finally {
    stream.close();
}
```

These are just two simple tests that show ways of testing a sequencer. Some tests may get quite involved, especially if a lot of output data is produced.

It may also be useful to create some integration tests that [configure JBoss DNA](#) to use a custom sequencer, and to then upload content using the JCR API, verifying that the custom sequencer did run. However, remember that JBoss DNA runs sequencers asynchronously in the background, and you must synchronize your tests to ensure that the sequencers have a chance to run before checking the results. (One way of doing this (although, granted, not always reliable) is to wait for a second after uploading your content, shutdown the

[SequencingService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html] and await its termination, and then check that the sequencer output has been saved to the JCR repository. For an example of this technique, see the `SequencingClientTest` unit test in the example application.)

7.4.4. Deploying custom sequencers

The first step of deploying a sequencer consists of adding/changing the sequencer configuration (e.g., [SequencerConfig](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencerConfig.html]) in the [SequencingService](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/sequencers/SequencingService.html]. This was covered in the [previous chapter](#).

The second step is to make the sequencer implementation available to JBoss DNA. At this time, the JAR containing your new sequencer, as well as any JARs that your sequencer depends on, should be placed on your application classpath.



Note

A future goal of JBoss DNA is to allow sequencers, connectors, and other extensions to be easily deployed into a runtime repository. This process will not only be much simpler, but it will also provide JBoss DNA with the information necessary to update configurations and create the appropriate class loaders for each extension. Having separate class loaders for each extension helps prevent the pollution of the common classpath, facilitates an isolated runtime environment to eliminate any dependency conflicts, and may potentially enable hot redeployment of newer extension versions.

7.5. Summary

In this chapter, we described how JBoss DNA sequences files as they're uploaded into a repository. And one of the things we mentioned was that each sequencer is handed (with other inputs) the MIME type of the file it is to process. How does DNA know what the MIME type is? JBoss DNA uses *MIME type detectors*, and this is the topic of the [next chapter](#).

MIME types

JBoss DNA often needs the ability to determine the MIME type for some binary content. When uploading content into a repository, we may want to add the MIME type as metadata. Or, we may want to make some processing decisions based upon the MIME type. So, JBoss DNA created a small pluggable framework for determining the MIME type by using the name of the file (e.g., extensions) and/or by reading the actual content. Technically, the framework delegates this to one or more extensions. And we've provided one extension that does a very good job at determining the MIME type from a large variety of file types. But if that isn't sufficient, you can always incorporate your own detector into JBoss DNA.

To use this system, you simply invoke a static method and supply the name of the content (e.g., the name of the file, with the extension) and the [InputStream](http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html) to the actual binary content:

```
MimeType.of(name,content)
```

The result is a [String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) containing the [MIME type](http://www.iana.org/assignments/media-types/) (e.g., "text/plain") or null if the MIME type cannot be determined. Note that the name or [InputStream](http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html) may be null, making this a very versatile utility.

8.1. JBoss DNA MIME type detectors

The principle component in this framework is the concept of a **detector**. A detector attempts to determine the MIME type using the name of the content (e.g., the file name) and the actual content itself. If the detector is able to determine the MIME type, it simply returns it as a string. If not, it merely returns null. Note, however, that a detector must be thread-safe.

Here is the interface:

```
package org.jboss.dna.graph.mimetype;
@ThreadSafe
public interface MimeTypeDetector {

    /**
     * Returns the MIME-type of a data source, using its supplied content and/or
     * its supplied name, depending upon the implementation. If the MIME-type
     * cannot be determined, either a "default" MIME-type or null may
```

```
* be returned, where the former will prevent earlier registered MIME-type
* detectors from being consulted.
*
* @param name The name of the data source; may be null.
* @param content The content of the data source; may be null.
* @return The MIME-type of the data source, or optionally null
* if the MIME-type could not be determined.
* @throws IOException [http://java.sun.com/j2se/1.5.0/docs/api/java/io/IOException.html] If
an error occurs reading the supplied content.
*/
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] mimeTypeOf( String [http://
java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] name,
InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html]
content ) throws IOException [http://java.sun.com/j2se/1.5.0/docs/api/java/io/IOException.html];
}
```

Detectors can be added to the [MimeType](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html] class using the `addDetector(MimeTypeDetectorConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html] config) method`, where the [MimeTypeDetectorConfig](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html] defines the name of the detector class, the name of the [class loader](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html), a name, and a description. It is also possible to set the [ClassLoaderFactory](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/common/component/ClassLoaderFactory.html] that the [MimeType](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html] singleton will use.

We'll next look at the MIME type detectors that are provided out of the box, and how to write your own.

8.1.1. Aperture MIME type detector

The [ApertureMimeTypeDetector](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/mimetype/ApertureMimeTypeDetector.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/mimetype/ApertureMimeTypeDetector.html] class is an implementation of [MimeTypeDetector](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html] that uses the [Aperture](http://aperture.sourceforge.net/) [http://aperture.sourceforge.net/] open-source library, which is a very capable utility for determining the MIME type for a wide range of file types, using both the file name and the actual content.

8.2. Writing custom detectors

Creating a custom detector involves the following steps:

- Create a Maven 2 project for your detector;
- Implement the [MimeTypeDetector](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html] interface with your own implementation, and create unit tests to verify the functionality and expected behavior;
- Add a [MimeTypeDetectorConfig](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html] to the [MimeType](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html] class in your application as described [earlier](#), and
- Deploy the JAR file with your implementation (as well as any dependencies), and make them available to JBoss DNA in your application.

It's that simple.

8.2.1. Creating the Maven 2 project

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already [set up to use Maven](#), using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.



Note

JBoss DNA may provide in the future a Maven archetype for creating detector projects. If you'd find this useful and would like to help create it, please [join the community](#).



Note

The **dna-mimetype-detector-aperture** project is a small, self-contained detector implementation that you can use to help you get going. Starting with this project's source and modifying it to suit your needs may be the easiest way to get started. See the subversion repository: <http://anonsvn.jboss.org/repos/dna/trunk/sequencers/dna-mimetype-detector-aperture/> [http://anonsvn.jboss.org/repos/dna/trunk/extensions/dna-mimetype-detector-aperture/]

You can create your Maven project any way you'd like. For examples, see the [Maven 2 documentation](http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project) [http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-common</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-graph</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</dependency>
```

These are minimum dependencies required for compiling a detector. Of course, you'll have to add other dependencies that your sequencer needs.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.4.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
```

```
<scope>test</scope>
</dependency>
```

After you've created the project, simply implement the [MimeTypeDetector](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html] interface. And testing should be quite straightforward, MIME type detectors don't require any other components. In your tests, simply instantiate your [MimeTypeDetector](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/graph/mimetype/MimeTypeDetector.html] implementation, supply various combinations of names and/or [InputStream](http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html) [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html]s, and verify the output is what you expect.

To use in your application, create a [MimeTypeDetectorConfig](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html] object with the name, description, and class information for your detector, and add to the [MimeType](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html] class using the `addDetector(MimeTypeDetectorConfig [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeTypeDetectorConfig.html] config) method`. Then, just use the [MimeType](http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html) [http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/apidocs/org/jboss/dna/repository/mimetype/MimeType.html] class.

Configuration

Configuring JBoss DNA services is a bit more manual than is ideal. JBoss DNA currently consists of classes that uses dependency injection to allow a great deal of flexibility in how it can be configured and customized. But this flexibility makes it more difficult for you to configure. We understand this, and will soon provide a much easier way to set up and manage JBoss DNA. Current plans are to use the *JBoss Microcontainer* [<http://www.jboss.org/jbossmc>] along with a configuration repository.

Testing

The JBoss DNA project uses automated testing to verify that the software is doing what it's supposed to and not doing what it shouldn't do. These automated tests are run continuously and also act as regression tests, ensuring that we know if any problems we find and fix reappear later. All of our tests are executed as part of our [Maven](#) build process, and the entire build process (including the tests) is automatically run using [Hudson](#) continuous integration system.

10.1. Unit tests

Unit tests verify the behavior of a single class (or small set of classes) in isolation from other classes. We use the JUnit 4.4 testing framework, which has significant improvements over earlier versions and makes it very easy to quickly write unit tests with little extra code. We also frequently use the Mockito library to help create mock implementations of other classes that are not under test but are used in the tests.

Unit tests should generally run quickly and should not require large assemblies of components. Additionally, they may rely upon the file resources included in the project, but these tests should require no external resources (like databases or servers). Note that our unit tests are run during the "test" phase of the standard [Maven lifecycle](#) [<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>]. This means that they are executed against the raw .class files created during compilation.

Developers are expected to run all of the JBoss DNA unit tests in their local environment before committing changes to SVN. So, if you're a developer and you've made changes to your local copy of the source, you can run those tests that are related to your changes using your IDE or with Maven (or any other mechanism). But before you commit your changes, you are expected to run a full Maven build using `mvn clean install` (in the "trunk/" directory). Please do *not* rely upon continuous integration to run all of the tests for you - the CI system is there to catch the occasional mistakes and to also run the [integration tests](#).

10.2. Integration tests

While [unit tests](#) test individual classes in (relative) isolation, the purpose of **integration tests** are to verify that assemblies of classes and components are behaving correctly. These assemblies are often the same ones that end users will actually use. In fact, integration tests are executed during the "integration-test" phase of the standard [Maven lifecycle](#) [<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>], meaning they are executed against the packaged JARs and artifacts of the project.

Integration tests also use the JUnit 4.4 framework, so they are again easy to write and follow the same pattern as unit tests. However, because they're working with larger assemblies of components, they often will take longer to set up, longer to run, and longer to tear down. They also may require initializing "external resources", like databases or servers.

Note, that while external resources may be required, care should be taken to minimize these dependencies and to ensure that most (if not all) integration tests may be run by anyone who downloads the source code. This means that these external resources should be available and set up within the tests. For example, use in-memory databases where possible. Or, if a database is required, use an open-source database (e.g., MySQL or PostgreSQL). And when these external resources are not available, it should be obvious from the test class names and/or test method names that it involved an external resource (e.g., `"MySQLConnectorIntegrationTest.shouldFindNodeStoredInDatabase()"`).

10.3. Writing tests

As mentioned in [the introduction](#), the JBoss DNA project doesn't follow any one methodology or process. Instead, we simply have a goal that as much code as possible is tested to ensure it behaves as expected. Do we expect 100% of the code is covered by automated tests? No, but we do want to test as much as we can. Maybe a simple JavaBean class doesn't need many tests, but any class with non-trivial logic should be tested.

We do encourage writing tests either before or while you write the code. Again, we're not blindly following a methodology. Instead, there's a very practical reason: writing the tests early on helps you write classes that are testable. If you wait until after the class (or classes) are done, you'll probably find that it's not easy to test all of the logic (especially the complicated logic).

Another suggestion is to write tests so that they specify and verify the behavior that is expected from a class or component. One challenge developers often have is knowing what they should even test and what the tests should look like. This is where [Behavior-driven development \(BDD\)](http://behaviour-driven.org/) [http://behaviour-driven.org/] helps out. If you think about what a class' behaviors are supposed to be (e.g., requirements), simply capture those requirements as test methods (with no implementations). For example, a test class for sequencer implementation might have a test method `shouldNotThrowAnErrorWhenTheSuppliedStreamIsNull()` { }. Then, after you enumerate all the requirements you can think of, go back and start implementing the test methods.

If you look at the existing test cases, you'll find that the names of the unit and integration tests in JBoss DNA follow a naming style, where the test method names are readable sentences. Actually, we try to name the test methods *and* the test classes such that they form a concisely-worded requirement. For example,

```
InMemorySequencerTest.shouldNotThrowAnErrorWhenTheSuppliedStreamIsNull()
```

is easily translated into a readable requirement:

```
InMemorySequencer should not throw an error when the supplied stream is null.
```


In fact, at some point in the future, we'd like to process the source to automatically generate a list of the behavior specifications that are asserted by the tests.

But for now, we write tests - a lot of them. And by following a few simple conventions and practices, we're able to do it quickly and in a way that makes it easy to understand what the code is supposed to do (or not do).

Looking to the future

What's next for JBoss DNA? Well, the sequencing system is just the beginning. With this release, the sequencing system is stable enough so that more [sequencers](#) can be developed and used within your own applications. We've also established the foundation for JBoss DNA repositories, including a number of [connectors](#). We'll continue to expand our library of sequencers and connectors, as well as expand our support of JCR. Other components on our roadmap include a web user interface, a REST-ful server, and a view system that allows domain-specific views of information in the repository. These components are farther out on our roadmap, and at this time have not been targeted to a particular release.

If you're interested in getting involved with the JBoss DNA project, consider picking up one of the sequencers on our [roadmap](#) [<http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel>]. Or, check out [JIRA](#) [<http://jira.jboss.org/jira/secure/IssueNavigator.jspa?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311436>] for the list of sequencers we've thought of. If you think of one that's not there, please add it to JIRA!

And, if you haven't already, check out our [Getting Started](#) [<http://www.jboss.org/file-access/default/members/dna/freezone/docs/0.3/manuals/gettingstarted/html/index.html>] guide, which has examples that you can build and run to see JBoss DNA in action.

